

Manual Programación Android

Salvador Gómez Oliver

WWW.SGOLIVER.NET

Versión 3.0

Versión 3.0 // Junio 2013

Este curso también está disponible **online**.

Es posible que exista una versión más reciente de este documento o que puedas encontrar contenido web actualizado.

Para más información te recomiendo que visites la web oficial del curso:

http://www.sgoliver.net/blog/?page_id=2935

INDICE DE CONTENIDOS

VI. Bases de Datos

VII. Preferencias en Android

VIII. Localización Geográfica

IX. Mapas en Android

X. Ficheros en Android

XI. Content Providers

XII. Notificaciones Android

XIII. Tareas en Segundo Plano

XIV. Acceso a Servicios Web

XV. Notificaciones Push

XVI. Depuración en Android

PRÓLOGO

Hay proyectos que se comienzan sin saber muy bien el rumbo exacto que se tomará, ni el destino que se pretende alcanzar. Proyectos cuyo único impulso es el día a día, sin planes, sin reglas, tan solo con el entusiasmo de seguir adelante, a veces con ganas, a veces sin fuerzas, pero siempre con la intuición de que va a salir bien.

El papel bajo estas líneas es uno de esos proyectos. Nació casi de la casualidad allá por 2010. Hoy, varios años después, sigue más vivo que nunca.

A pesar de llevar metido en el desarrollo para Android casi desde sus inicios, en mi blog [sgoliver.net] nunca había tratado estos temas, pretendía mantenerme fiel a su temática original: el desarrollo bajo las plataformas Java y .NET. Surgieron en algún momento algunos escarceos con otros lenguajes, pero siempre con un ojo puesto en los dos primeros.

Mi formación en Android fue en inglés. No había alternativa, era el único idioma en el que, por aquel entonces, existía buena documentación sobre la plataforma. Desde el primer concepto hasta el último tuve que aprenderlo en el idioma de Shakespeare. A día de hoy esto no ha cambiado mucho, la buena documentación sobre Android, la buena de verdad, sigue y seguirá aún durante algún tiempo estando en inglés, pero afortunadamente son ya muchas las personas de habla hispana las que se están ocupando de ir equilibrando poco a poco esta balanza de idiomas.

Y con ese afán de aportar un pequeño granito de arena a la comunidad hispanohablante es como acabé decidiendo dar un giro, quien sabe si temporal o permanente, a mi blog y comenzar a escribir sobre desarrollo para la plataforma Android. No sabía hasta dónde iba a llegar, no sabía la aceptación que tendría, pero lo que sí sabía es que me apetecía ayudar un poco a los que como yo les costaba encontrar información básica sobre Android disponible en su idioma.

Hoy, gracias a todo vuestro apoyo, vuestra colaboración, vuestras propuestas, y vuestras críticas (de todo se aprende) éste es un proyecto con varios años ya de vida. Más de 300 páginas, más de 50 artículos, y sobre todo cientos de comentarios de ánimo recibidos.

Y este documento no es un final, es sólo un punto y seguido. Este libro es tan solo la mejor forma que he encontrado de mirar atrás, ordenar ideas, y pensar en el siguiente camino a tomar, que espero sea largo. Espero que muchos de vosotros me acompañéis en parte de ese camino igual que lo habéis hecho en el recorrido hasta ahora.

Muchas gracias, y que comience el espectáculo.

¿A QUIÉN VA DIRIGIDO ESTE LIBRO?

Este manual va dirigido a todas aquellas personas interesadas en un tema tan en auge como la programación de aplicaciones móviles para la plataforma Android. Se tratarán temas dedicados a la construcción de aplicaciones nativas de la plataforma, dejando a un lado por el momento las aplicaciones web. Es por ello por lo que el único requisito indispensable a la hora de utilizar este manual es tener conocimientos bien asentados sobre el lenguaje de programación Java y ciertas nociones sobre aspectos básicos del desarrollo actual como la orientación a objetos.

LICENCIA

1

Conceptos Básicos

I. Conceptos Básicos

Entorno de desarrollo Android

En este apartado vamos a describir los pasos básicos para disponer en nuestro PC del entorno y las herramientas necesarias para comenzar a programar aplicaciones para la plataforma Android.

No voy a ser exhaustivo, ya existen muy buenos tutoriales sobre la instalación de Eclipse y Android, incluida la documentación oficial de la plataforma. Además, si has llegado hasta aquí quiero suponer que tienes unos conocimientos básicos de Eclipse y Java, por lo que tan sólo enumeraré los pasos necesarios de instalación y configuración, y proporcionaré los enlaces a las distintas herramientas. Vamos allá.

Paso 1. Descarga e instalación de Java.

Si aún no tienes instalado ninguna versión del JDK (*Java Development Kit*) puedes descargar la última versión desde la web de Oracle.



En el momento de escribir este manual la versión más reciente disponible es la 7 update7, que podremos descargar para nuestra versión del sistema operativo, en mi caso la versión para Windows 64 bits.

Product / File Description	File Size	Download
Linux x86	120.62 MB	jdk-7u7-linux-i586.rpm
Linux x86	92.86 MB	jdk-7u7-linux-i586.tar.gz
Linux x64	118.8 MB	jdk-7u7-linux-x64.rpm
Linux x64	91.59 MB	jdk-7u7-linux-x64.tar.gz
Mac OS X	143.46 MB	jdk-7u7-macosx-x64.dmg
Solaris x86	135.4 MB	jdk-7u7-solaris-i586.tar.Z
Solaris x86	91.86 MB	jdk-7u7-solaris-i586.tar.gz
Solaris x64	22.51 MB	jdk-7u7-solaris-x64.tar.Z
Solaris x64	14.95 MB	jdk-7u7-solaris-x64.tar.gz
Solaris SPARC	135.69 MB	jdk-7u7-solaris-sparc.tar.Z
Solaris SPARC	95.15 MB	jdk-7u7-solaris-sparc.tar.gz
Solaris SPARC 64-bit	22.75 MB	jdk-7u7-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	17.47 MB	jdk-7u7-solaris-sparcv9.tar.gz
Windows x86	88.36 MB	jdk-7u7-windows-i586.exe
Windows x64	90 MB	jdk-7u7-windows-x64.exe

La instalación no tiene ninguna dificultad ya que es un instalador estándar de Windows donde tan sólo hay que aceptar las opciones que ofrece por defecto.

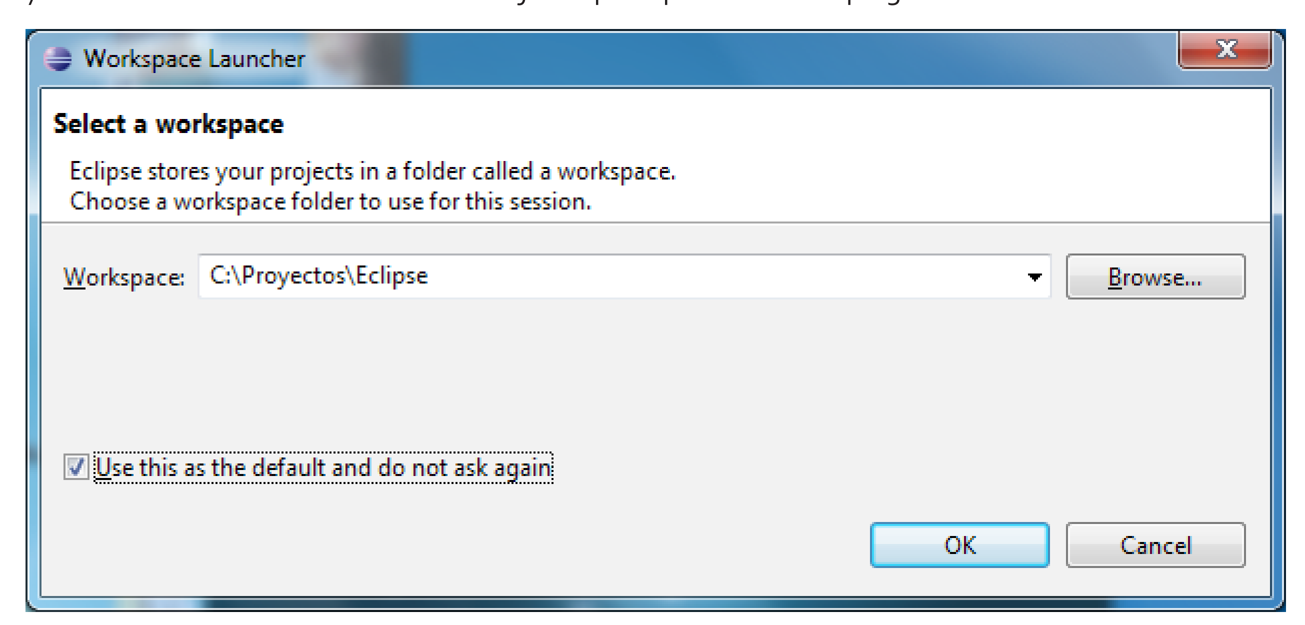
Paso 2. Descarga e instalación de Eclipse.

Si aún no tienes instalado Eclipse, puedes descargar la última versión, la 4.2.1 [Eclipse Juno SR1] en la última revisión de este texto, desde este enlace. Recomiendo descargar la versión *Eclipse IDE for Java Developers*, y por supuesto descargar la versión apropiada para tu sistema operativo (Windows/Mac OS/Linux, y 32/64 bits). Durante el curso siempre utilizaré Windows 64 bits.





La instalación consiste simplemente en descomprimir el zip descargado en la ubicación deseada. Para ejecutarlo accederemos al fichero eclipse.exe dentro de la ruta donde hayamos descomprimido la aplicación, por ejemplo `c:\eclipse\eclipse.exe`. Durante la primera ejecución de la aplicación nos preguntará cuál será la carpeta donde queremos almacenar nuestros proyectos. Indicaremos la ruta deseada y marcaremos la check "*Use this as the default*" para que no vuelva a preguntarlo.



Paso 3. Descargar el SDK de Android.

El SDK de la plataforma Android se puede descargar desde aquí (en el momento de revisar este texto la última versión es la r21, que funciona perfectamente con Eclipse 4.2.1). Una vez descargado, bastará con ejecutar el instalador estándar de Windows.

USE AN EXISTING IDE

If you already have an IDE you want to use for Android app de
download the SDK Tools, then select additional Android SDK
system image). If you'll be using an existing version of Eclips

Download the SDK Tools for Windows

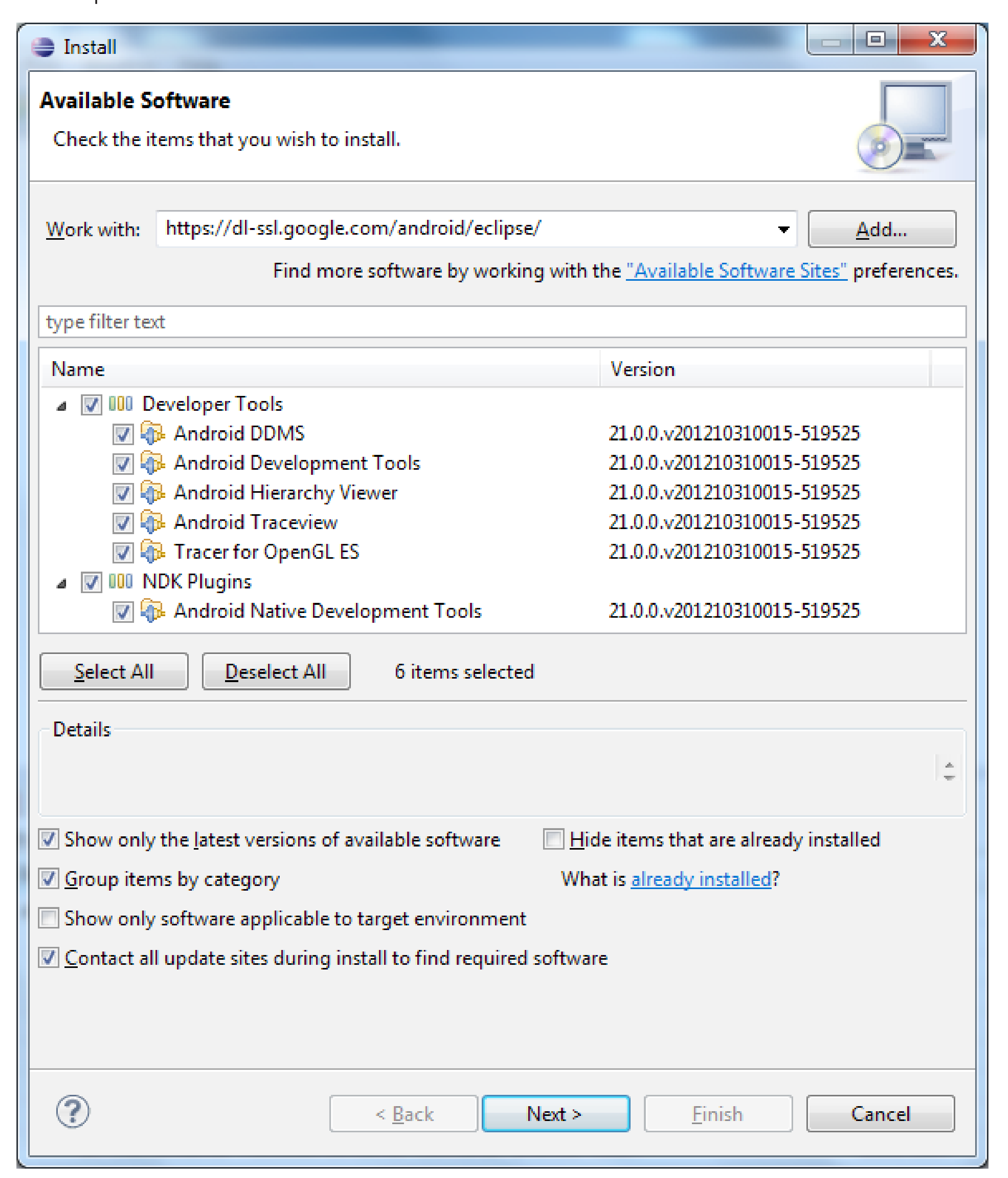
Paso 4. Descargar el plugin de Android para Eclipse.

Google pone a disposición de los desarrolladores un plugin para Eclipse llamado *Android Development Tools* (ADT) que facilita en gran medida el desarrollo de aplicaciones para la plataforma. Podéis descargarlo

mediante las opciones de actualización de Eclipse, accediendo al menú "*Help / Install new software...*" e indicando la siguiente URL de descarga:

```
https://dl-ssl.google.com/android/eclipse/
```

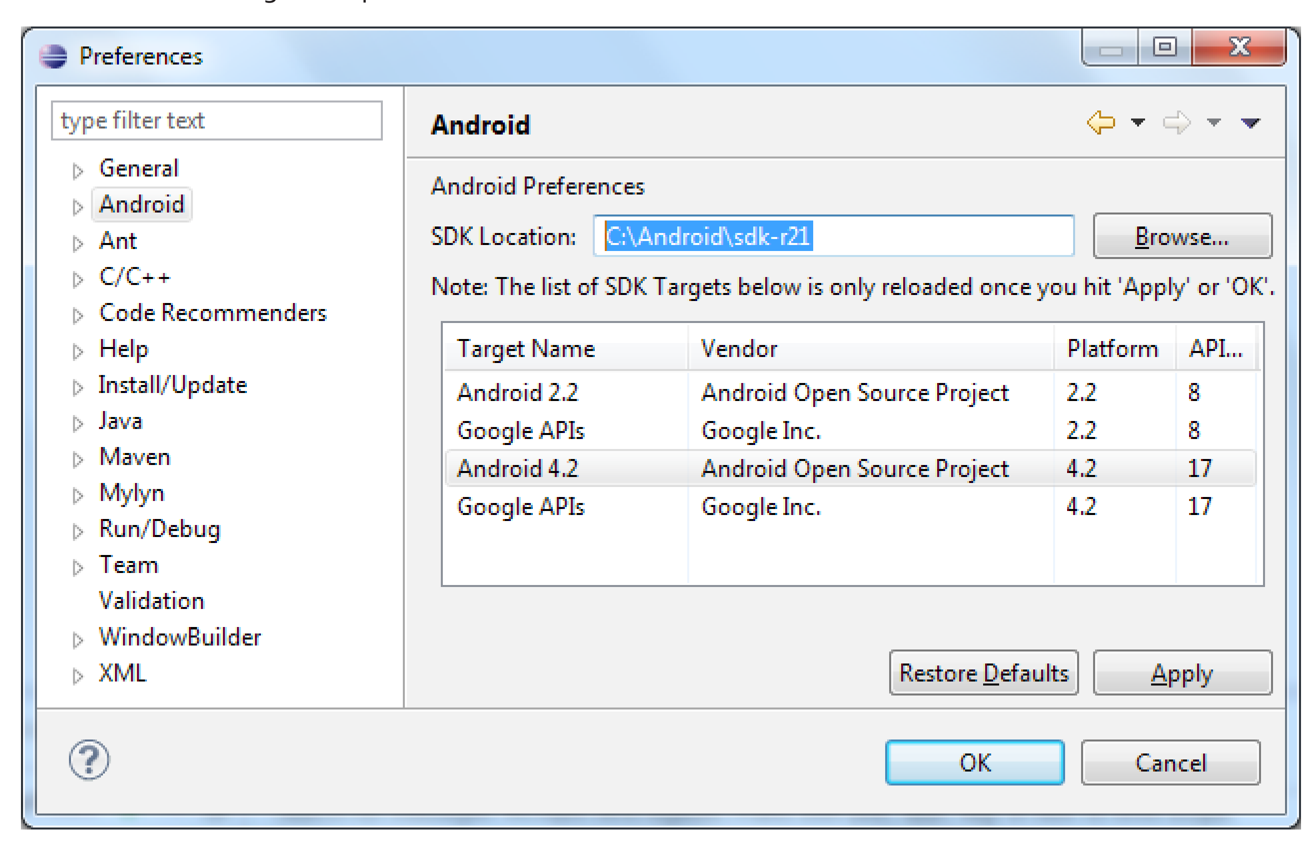
Seleccionaremos los dos paquetes disponibles "*Developer Tools*" y "*NDK Plugins*" y pulsaremos el botón "*Next>*" para comenzar con el asistente de instalación.



Durante la instalación Eclipse te pedirá que aceptes la licencia de los componentes de Google que vas a instalar y es posible que aparezca algún mensaje de *warning* que simplemente puedes aceptar para continuar con la instalación. Finalmente el instalador te pedirá que reinicies Eclipse.

Paso 5. Configurar el plugin ADT.

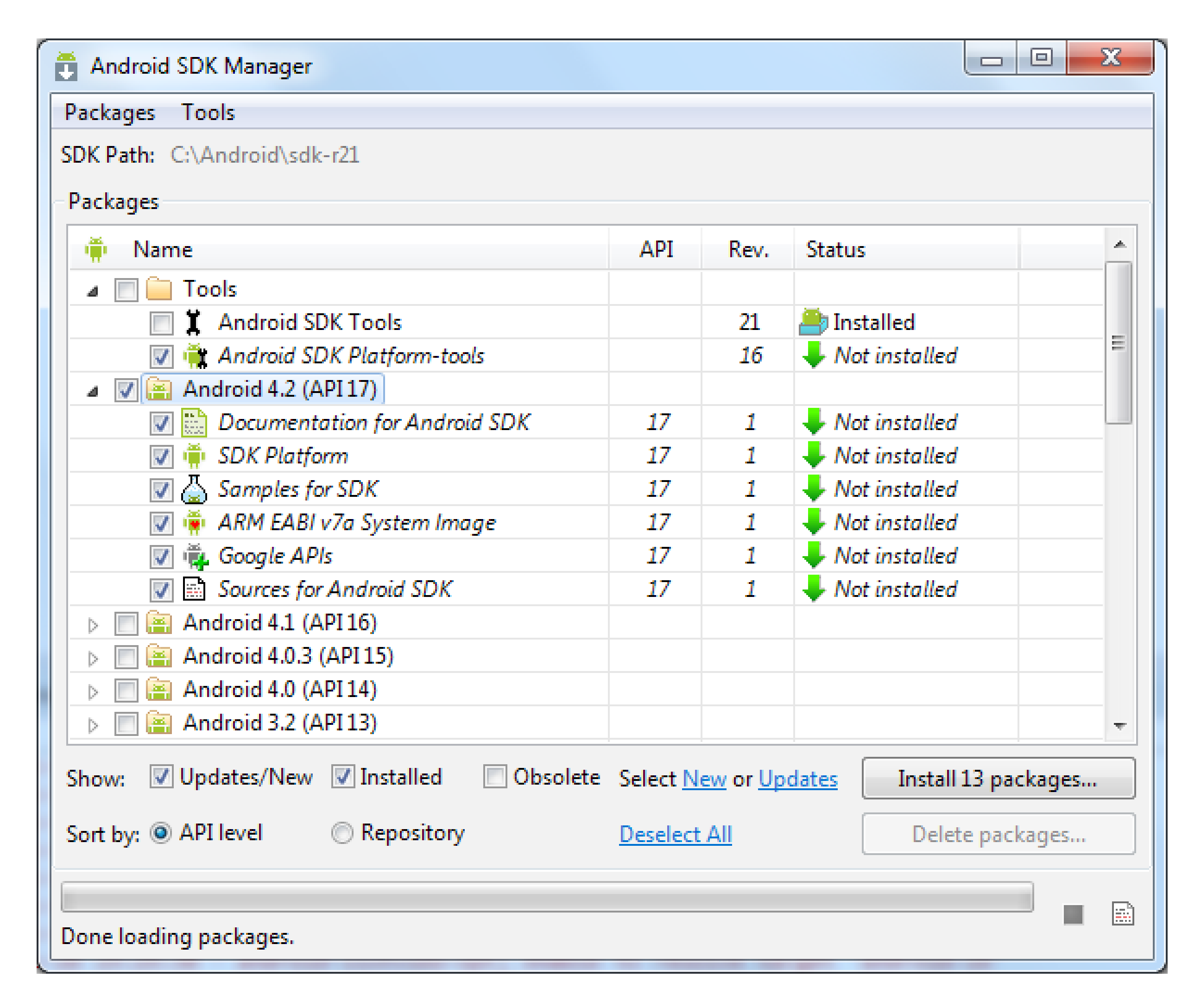
Una vez instalado el *plugin*, tendremos que configurarlo indicando la ruta en la que hemos instalado el SDK de Android. Para ello, iremos a la ventana de configuración de Eclipse (Window / Preferences...), y en la sección de Android indicaremos la ruta en la que se ha instalado. Finalmente pulsaremos OK para aceptar los cambios. Si aparece algún mensaje de *warning* aceptamos sin más, ya que se son problemas que se solucionarán en el siguiente paso.



Paso 6. Instalar las Platform Tools y los Platforms necesarios.

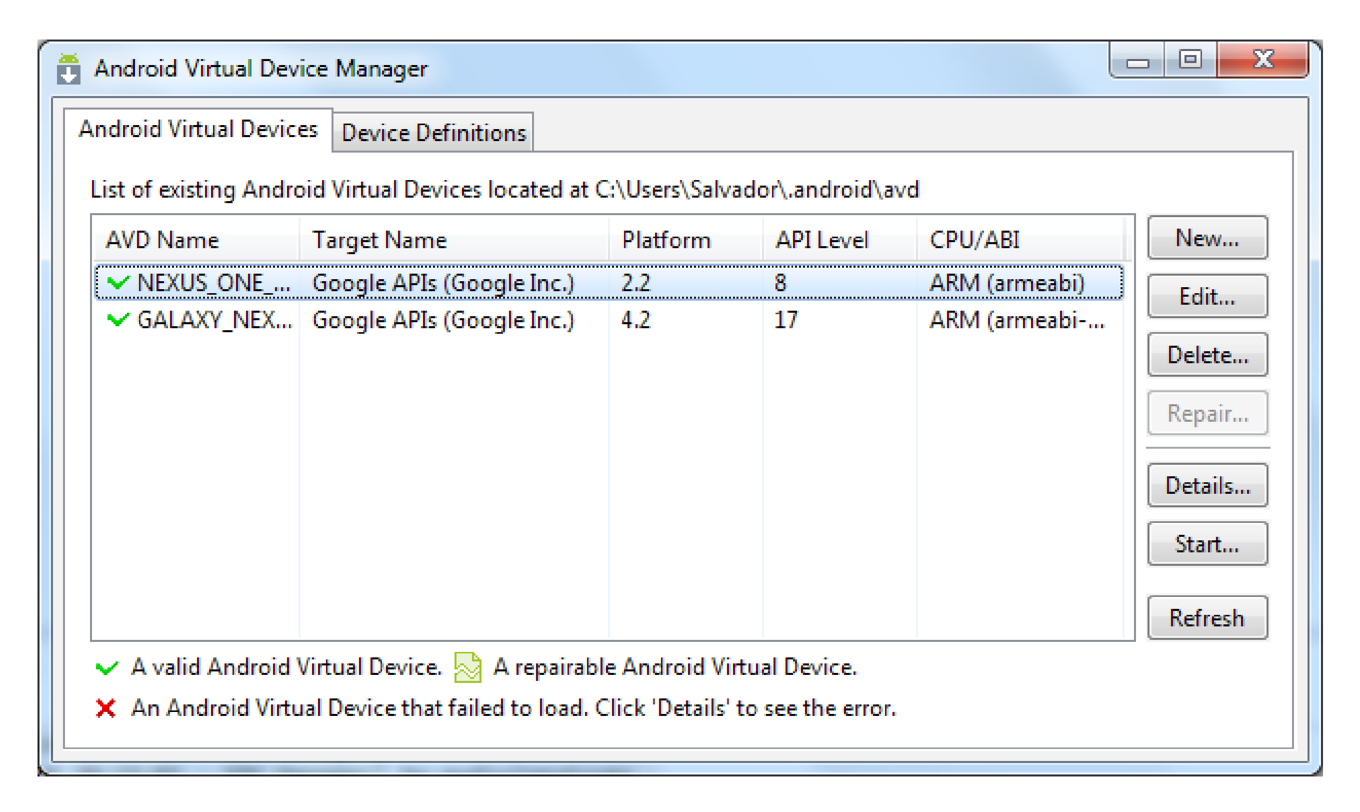
Además del SDK de Android comentado en el paso 2, que contiene las herramientas básicas para desarrollar en Android, también deberemos descargar las llamadas *Platflorm Tools*, que contiene herramientas específicas de la última versión de la plataforma, y una o varias plataformas (*SDK Platforms*) de Android, que no son más que las librerías necesarias para desarrollar sobre cada una de las versiones concretas de Android. Así, si queremos desarrollar por ejemplo para Android 2.2 tendremos que descargar su plataforma correspondiente. Mi consejo personal es siempre instalar al menos 2 plataformas: la correspondiente a la última versión disponible de Android, y la correspondiente a la mínima versión de Android que queremos que soporte nuestra aplicación.

Para ello, desde Eclipse debemos acceder al menú "*Window / Android SDK Manager*". En la lista de paquetes disponibles seleccionaremos las "Android SDK Platform-tools", las plataformas "*Android 4.2 (API 17)*" y "*Android 2.2 (API 8)*", y el paquete extra "*Android Support Library*", que es una librería que nos permitirá utilizar en versiones antiguas de Android características introducidas por versiones más recientes. Pulsaremos el botón "*Install packages...*" y esperaremos a que finalice la descarga.

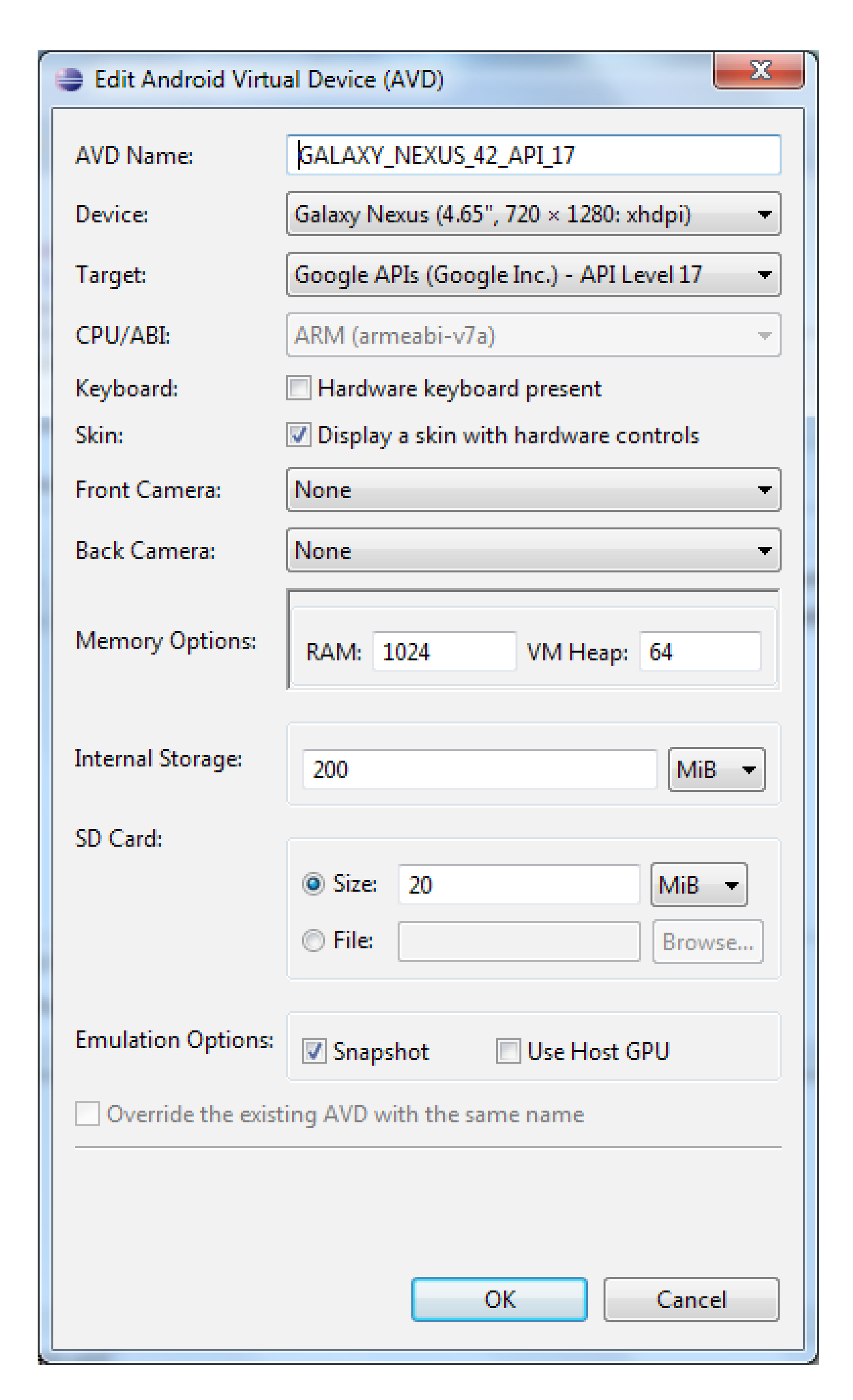


Paso 7. Configurar un AVD.

A la hora de probar y depurar aplicaciones Android no tendremos que hacerlo necesariamente sobre un dispositivo físico, sino que podremos configurar un emulador o dispositivo virtual (*Android Virtual Device*, o AVD) donde poder realizar fácilmente estas tareas. Para ello, accederemos al AVD Manager (menú Window / AVD Manager), y en la sección *Virtual Devices* podremos añadir tantos AVD como se necesiten (por ejemplo, configurados para distintas versiones de Android o distintos tipos de dispositivo). Nuevamente, mi consejo será configurar al menos dos AVD, uno para la mínima versión de Android que queramos soportar, y otro para la versión más reciente disponible.



Para configurar el AVD tan sólo tendremos que indicar un nombre descriptivo, la versión de la plataforma Android que utilizará, y las características de hardware del dispositivo virtual, como por ejemplo su resolución de pantalla o el tamaño de la tarjeta SD. Además, marcaremos la opción "*Snapshot*", que nos permitirá arrancar el emulador más rápidamente en futuras ejecuciones.

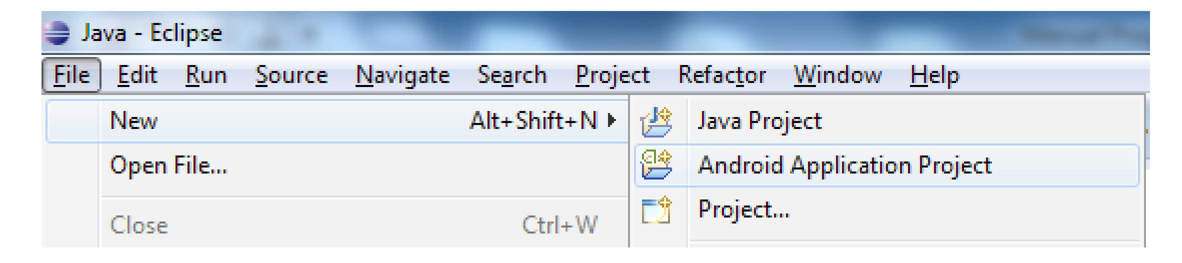


Y con este paso ya tendríamos preparadas todas las herramientas necesarias para comenzar a desarrollar aplicaciones Android. En próximos apartados veremos como crear un nuevo proyecto, la estructura y componentes de un proyecto Android, y crearemos una aplicación sencilla para poner en práctica todos los conceptos aprendidos.

Estructura de un proyecto Android

Para empezar a comprender cómo se construye una aplicación Android vamos a crear un nuevo proyecto Android en Eclipse y echaremos un vistazo a la estructura general del proyecto creado por defecto.

Para crear un nuevo proyecto abriremos Eclipse e iremos al menú File / New / Android Application Project.



De esta forma iniciaremos el asistente de creación del proyecto, que nos guiará por las distintas opciones de creación y configuración de un nuevo proyecto.

En la primera pantalla indicaremos el nombre de la aplicación, el nombre del proyecto y el paquete java que utilizaremos en nuestras clases java. Tendremos que seleccionar además la mínima versión del SDK que aceptará nuestra aplicación al ser instalada en un dispositivo (*Minimum Required SDK*), la versión del SDK para la que desarrollaremos (*Target SDK*), y la versión del SDK con la que compilaremos el proyecto (*Compile with*). Las dos últimas suelen coincidir con la versión de Android más reciente. El resto de opciones las dejaremos con los valores por defecto.

New Android Application

New Android Application

Creates a new Android Application

Application Name: Hola Usuario

Project Name: android-hola-usuario

Package Name: net.sgoliver.android.holausuario

Minimum Required SDK: API 8: Android 2.2 (Froyo)

Target SDK: API 17: Android 4.2

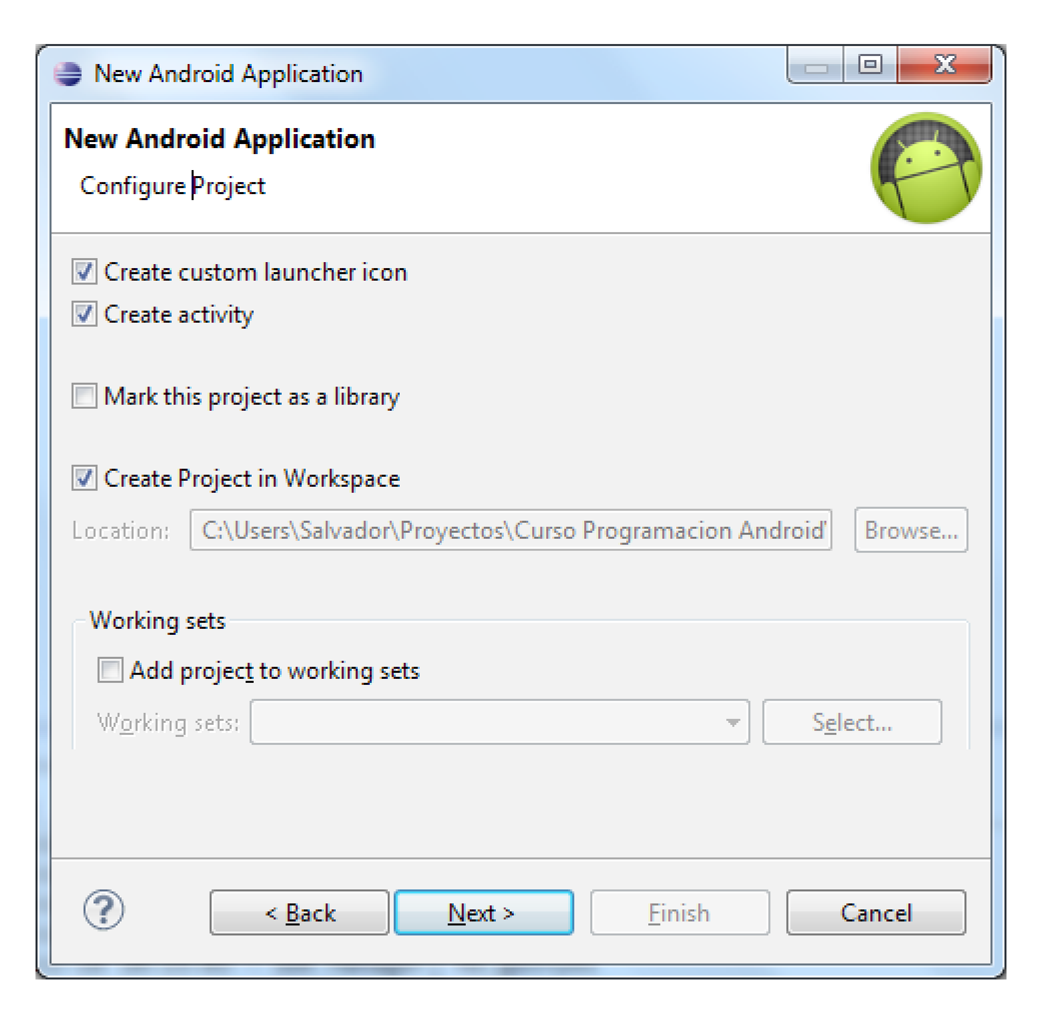
Compile With: API 17: Android 4.2

Theme: Holo Light with Dark Action Bar

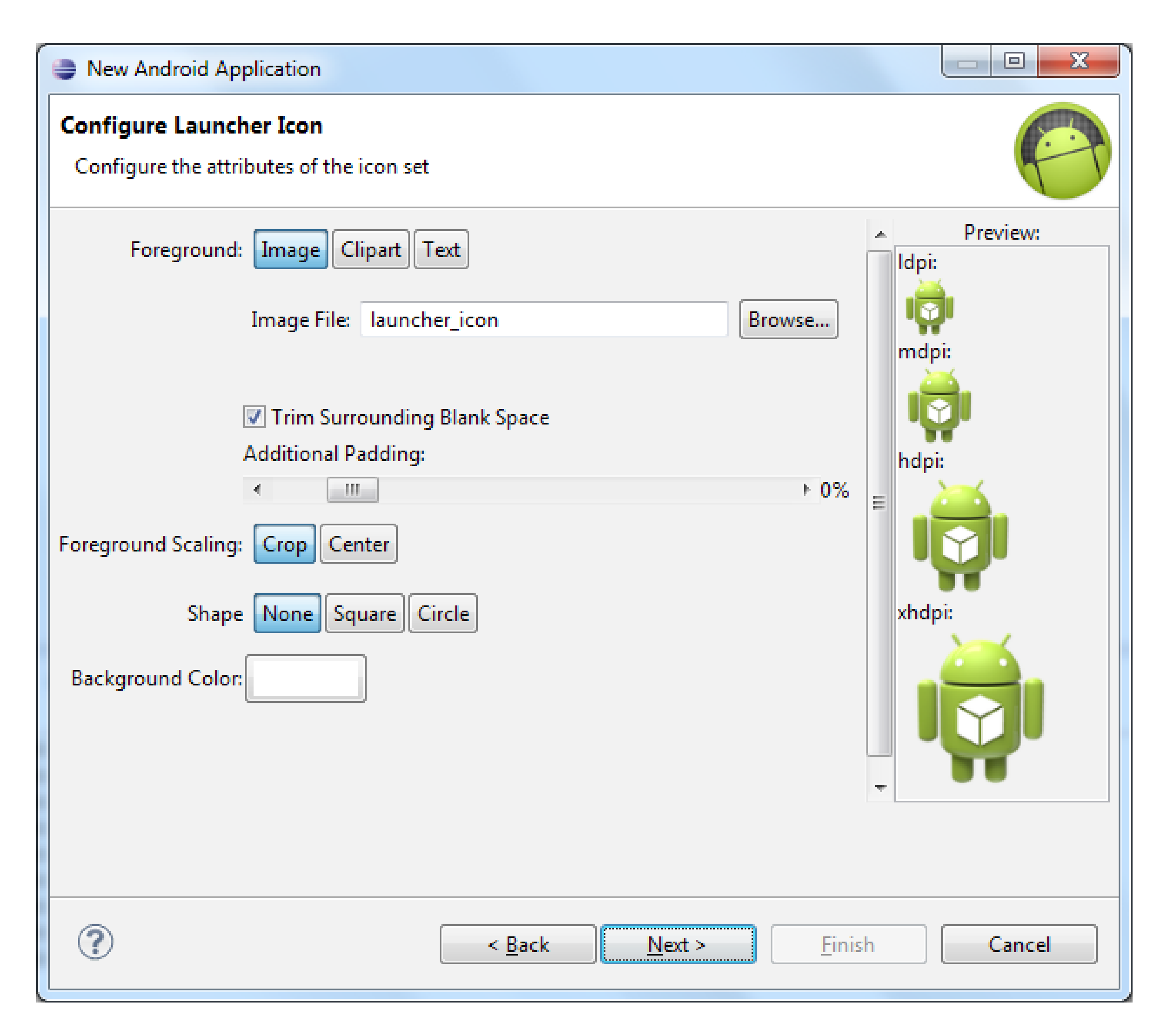
The application name is shown in the Play Store, as well as in the Manage Application list in Settings.

< Back | Next > | Finish | Cancel

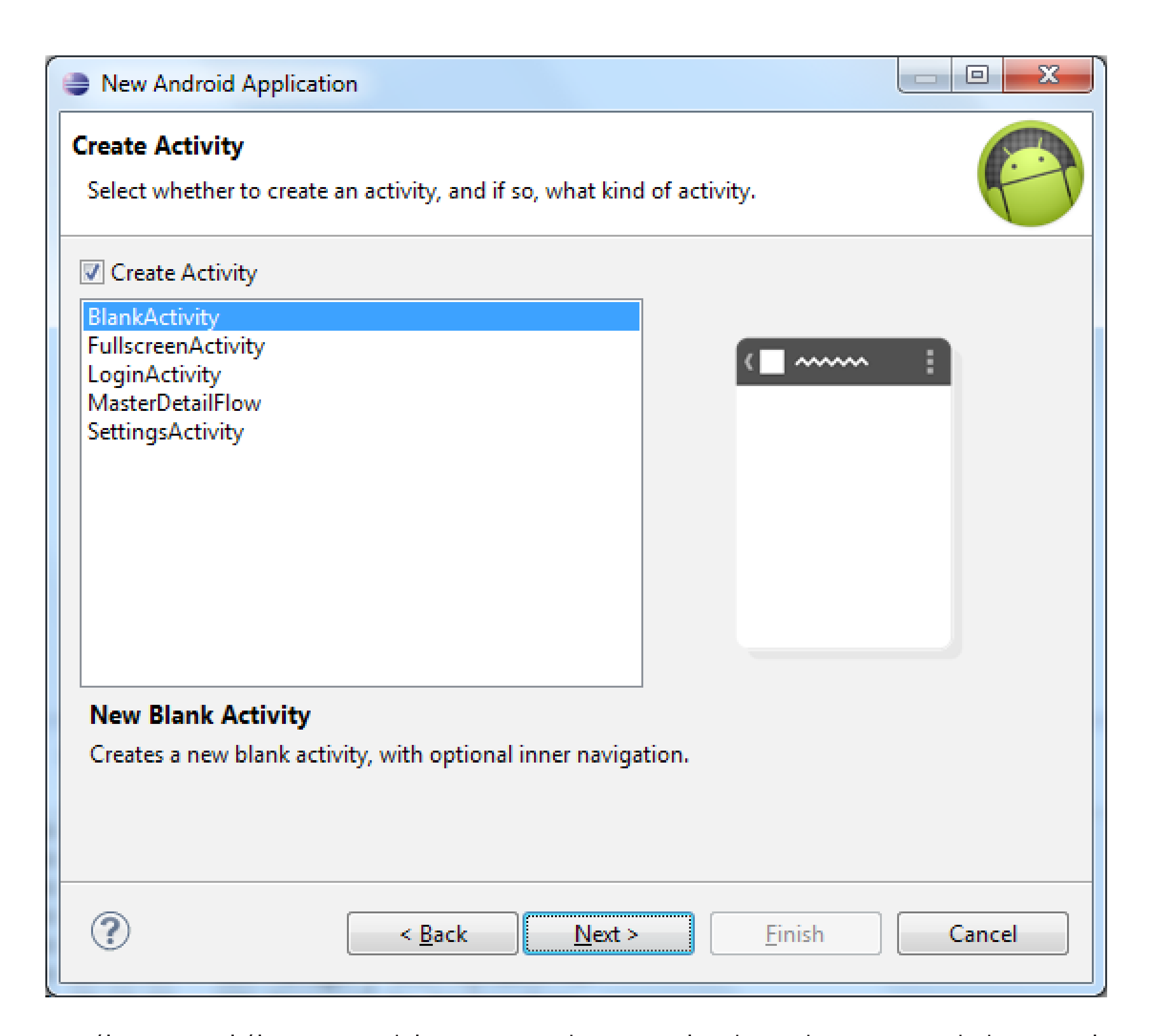
Al pulsar el botón *Next*, accederemos al segundo paso del asistente, donde tendremos que indicar si durante la creación del nuevo proyecto queremos crear un icono para nuestra aplicación (*Create custom launcher icon*) y si queremos crear una actividad inicial (*Create activity*). También podremos indicar si nuestro proyecto será del tipo Librería (*Mark this Project as a library*). Por ahora dejaremos todas las opciones marcadas por defecto como se ve en la siguiente imagen y pulsamos *Next*.



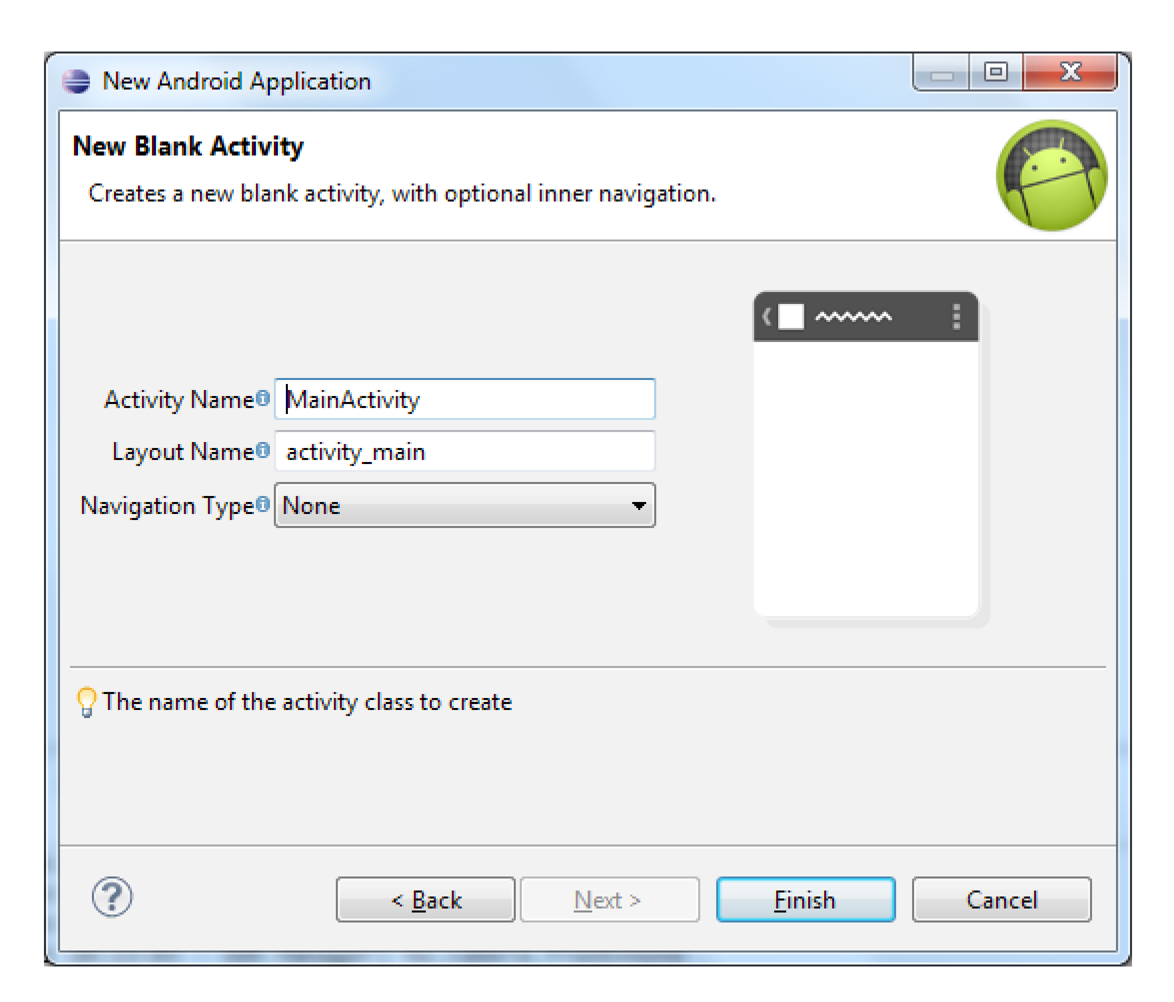
En la siguiente pantalla del asistente configuraremos el icono que tendrá nuestra aplicación en el dispositivo. No nos detendremos mucho en este paso ya que no tiene demasiada relevancia por el momento. Tan sólo decir que podremos seleccionar la imagen, texto o dibujo predefinido que aparecerá en el icono, el margen, la forma y los colores aplicados. Por ahora podemos dejarlo todo por defecto y avanzar al siguiente paso pulsando *Next*.



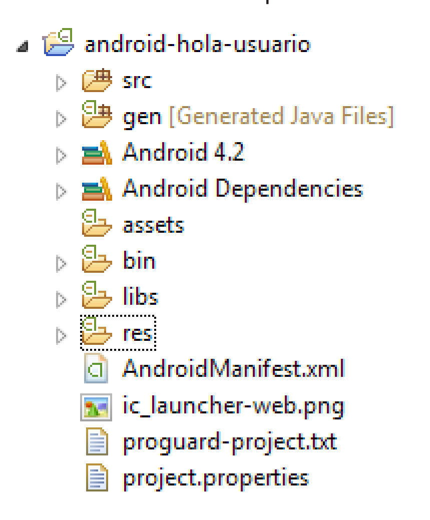
En la siguiente pantalla del asistente elegiremos el tipo de Actividad principal de la aplicación. Entenderemos por ahora que una *actividad* es una "ventana" o "pantalla" de la aplicación. En este paso también dejaremos todos los valores por defecto, indicando así que nuestra pantalla principal será del tipo *BlankActivity*.



Por último, en el último paso del asistente indicaremos los datos de esta actividad principal que acabamos de elegir, indicando el nombre de su clase java asociada y el nombre de su *layout xml* (algo así como la interfaz gráfica de la actividad, lo veremos más adelante).



Una vez configurado todo pulsamos el botón *Finish* y Eclipse creará por nosotros toda la estructura del proyecto y los elementos indispensables que debe contener. En la siguiente imagen vemos los elementos creados inicialmente para un nuevo proyecto Android:



En los siguientes apartados describiremos los elementos principales de esta estructura.

Carpeta /src/

Esta carpeta contendrá todo el código fuente de la aplicación, código de la interfaz gráfica, clases auxiliares, etc. Inicialmente, Eclipse creará por nosotros el código básico de la pantalla (`Activity`) principal de la aplicación, que recordemos que en nuestro caso era `MainActivity`, y siempre bajo la estructura del paquete java definido.

src
net.sgoliver.android.holausuario
MainActivity.java
MainActivity

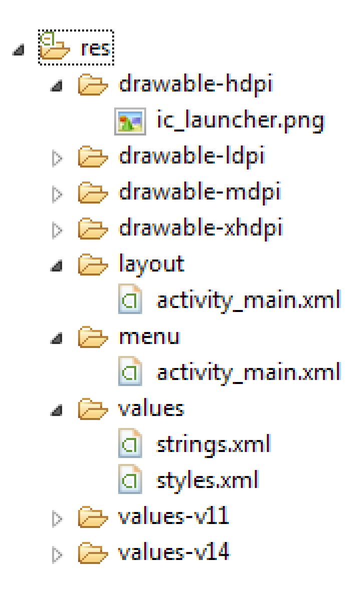
Carpeta /res/

Contiene todos los ficheros de recursos necesarios para el proyecto: imágenes, vídeos, cadenas de texto, etc. Los diferentes tipos de recursos se distribuyen entre las siguientes subcarpetas:

Carpeta	Descripción
`/res/drawable/`	Contiene las imágenes [y otros elementos gráficos] usados en por la aplicación. Para definir diferentes recursos dependiendo de la resolución y densidad de la pantalla del dispositivo se suele dividir en varias subcarpetas: ➢ `/drawable-ldpi` (densidad baja) ➢ `/drawable-mdpi` (densidad media) ➢ `/drawable-hdpi` (densidad alta) ➢ `/drawable-xhdpi` (densidad muy alta)
`/res/layout/`	Contiene los ficheros de definición XML de las diferentes pantallas de la interfaz gráfica. Para definir distintos *layouts* dependiendo de la orientación del dispositivo se puede dividir en dos subcarpetas: ➢ `/layout` (vertical) ➢ `/layout-land` (horizontal)
`/res/anim/` `/res/animator/`	Contienen la definición de las animaciones utilizadas por la aplicación.
`/res/color/`	Contiene ficheros XML de definición de colores según estado.
`/res/menu/`	Contiene la definición XML de los menús de la aplicación.
`/res/values/`	Contiene otros ficheros XML de recursos de la aplicación, como por ejemplo cadenas de texto (*strings.xml*), estilos (*styles.xml*), colores (*colors.xml*), arrays de valores (*arrays.xml*), etc.
`/res/xml/`	Contiene otros ficheros XML de datos utilizados por la aplicación.
`/res/raw/`	Contiene recursos adicionales, normalmente en formato distinto a XML, que no se incluyan en el resto de carpetas de recursos.

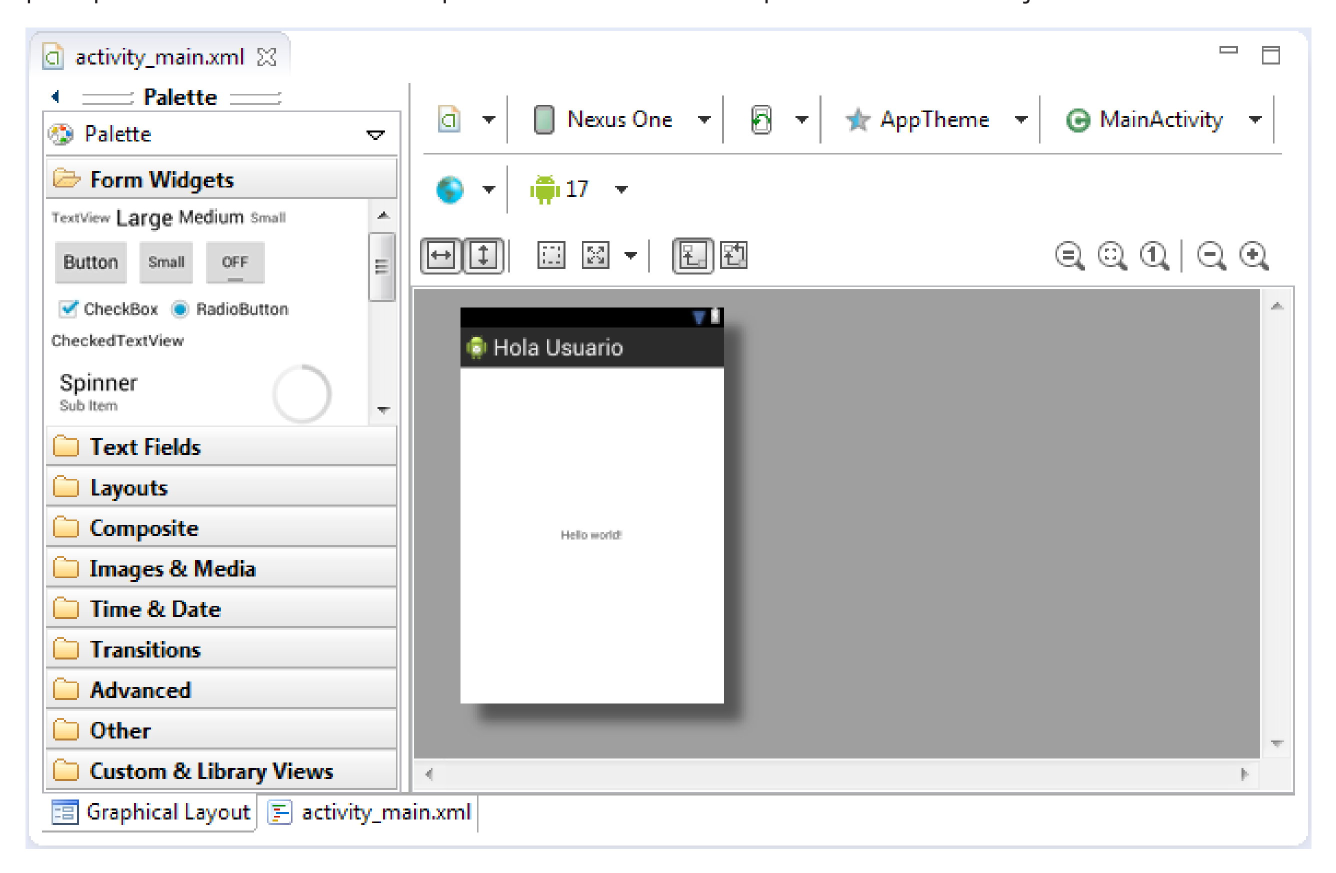
No todas estas carpetas tienen por qué aparecer en cada proyecto Android, tan sólo las que se necesiten. Iremos viendo durante el curso que tipo de elementos se pueden incluir en cada una de estas carpetas.

Como ejemplo, para un proyecto nuevo Android, se crean por defecto los siguientes recursos para la aplicación:



Como se puede observar, existen algunas carpetas en cuyo nombre se incluye un sufijo adicional, como por ejemplo "values-v11" y "values-v14". Estos, y otros sufijos, se emplean para definir recursos independientes para determinados dispositivos según sus características. De esta forma, por ejemplo, los recursos incluidos en la carpeta "values-v11" se aplicarían tan sólo a dispositivos cuya versión de Android sea la 3.0 (API 11) o superior. Al igual que el sufijo "–v" existen otros muchos para referirse a otras características del terminal, puede consultarse la lista completa en la documentación oficial del Android.

Entre los recursos creados por defecto, cabe destacar el layout "*activity_main.xml*", que contiene la definición de la interfaz gráfica de la pantalla principal de la aplicación. Si hacemos doble clic sobre el fichero Eclipse nos mostrará esta interfaz en su editor gráfico (tipo arrastrar y soltar) y como podremos comprobar, en principio contiene tan sólo una etiqueta de texto centrada en pantalla con el mensaje "Hello World!".



Durante el curso no utilizaremos demasiado este editor gráfico, sino que modificaremos la interfaz de nuestras pantallas manipulando directamente el fichero XML asociado (al que se puede acceder pulsando

sobre la pestaña inferior derecha, junto la solapa "*Graphical Layout*" que se observa en la imagen. En este caso, el XML asociado sería el siguiente:

```
activity_main.xml

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" >

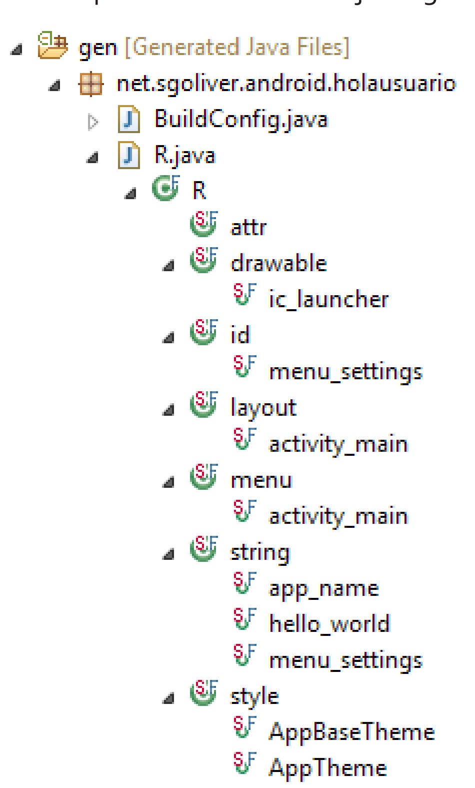
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/hello_world" />

</RelativeLayout>
```

Esto en principio puede parecer mucho más complicado que utilizar el editor gráfico, pero por el contrario [además de no ser nada complicado en realidad] nos permitirá aprender los entresijos de Android más rápidamente.

Carpeta /gen/

Contiene una serie de elementos de código **generados automáticamente** al compilar el proyecto. Cada vez que generamos nuestro proyecto, la maquinaria de compilación de Android genera por nosotros una serie de ficheros fuente java dirigidos al control de los recursos de la aplicación. Importante: dado que estos ficheros se generan automáticamente tras cada compilación del proyecto es importante que no se modifiquen manualmente bajo ninguna circunstancia.



A destacar sobre todo el fichero que aparece desplegado en la imagen anterior, llamado `R.java`, donde se define la clase `R`.

Esta clase `R` contendrá en todo momento una serie de constantes con los ID de todos los recursos de la aplicación incluidos en la carpeta `/res/`, de forma que podamos acceder fácilmente a estos recursos desde nuestro código a través de este dato. Así, por ejemplo, la constante `R.drawable.ic_launcher` contendrá el ID de la imagen "*ic_launcher.png*" contenida en la carpeta `/res/drawable/`. Veamos como ejemplo la clase R creada por defecto para un proyecto nuevo:

```
package net.sgoliver.android.holausuario;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int ic_launcher=0x7f020001;
    }
    public static final class id {
        public static final int menu_settings=0x7f070000;
    }
    public static final class layout {
        public static final int activity_main=0x7f030000;
    }
    public static final class menu {
        public static final int activity_main=0x7f060000;
    }
    public static final class string {
        public static final int app_name=0x7f040000;
        public static final int hello_world=0x7f040001;
        public static final int menu_settings=0x7f040002;
    }
    public static final class style {
        /**
        Base application theme, dependent on API level. This theme is replaced
        by AppBaseTheme from res/values-vXX/styles.xml on newer devices.


            Theme customizations available in newer API levels can go in
            res/values-vXX/styles.xml, while customizations related to
            backward-compatibility can go here.

        Base application theme for API 11+. This theme completely replaces
        AppBaseTheme from res/values/styles.xml on API 11+ devices.

 API 11 theme customizations can go here.

        Base application theme for API 14+. This theme completely replaces
        AppBaseTheme from BOTH res/values/styles.xml and
        res/values-v11/styles.xml on API 14+ devices.

 API 14 theme customizations can go here.
         */
        public static final int AppBaseTheme=0x7f050000;
        /**  Application theme.
 All customizations that are NOT specific to a particular API-level can go
here.
         */
        public static final int AppTheme=0x7f050001;
    }
}
```

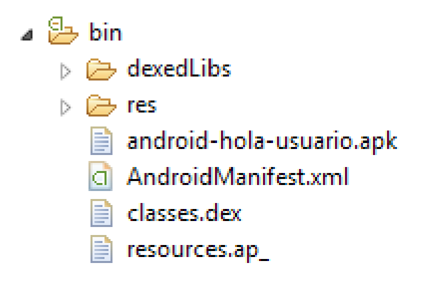
Carpeta /assets/

Contiene todos los demás ficheros auxiliares necesarios para la aplicación (y que se incluirán en su propio paquete), como por ejemplo ficheros de configuración, de datos, etc.

La diferencia entre los recursos incluidos en la carpeta `/res/raw/` y los incluidos en la carpeta `/assets/` es que para los primeros se generará un ID en la clase `R` y se deberá acceder a ellos con los diferentes métodos de acceso a recursos. Para los segundos sin embargo no se generarán ID y se podrá acceder a ellos por su ruta como a cualquier otro fichero del sistema. Usaremos uno u otro según las necesidades de nuestra aplicación.

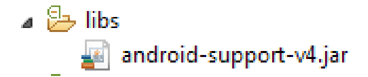
Carpeta /bin/

Ésta es otra de ésas carpetas que en principio no tendremos por qué tocar. Contiene los elementos compilados de la aplicación y otros ficheros auxiliares. Cabe destacar el fichero con extensión ".apk", que es el ejecutable de la aplicación que se instalará en el dispositivo.



Carpeta /libs/

Contendrá las librerías auxiliares, normalmente en formato ".jar" que utilicemos en nuestra aplicación Android.



Fichero AndroidManifest.xml

Contiene la definición en XML de los aspectos principales de la aplicación, como por ejemplo su identificación (nombre, versión, icono, ...), sus componentes (pantallas, mensajes, ...), las librerías auxiliares utilizadas, o los permisos necesarios para su ejecución. Veremos más adelante más detalles de este fichero.

Y con esto todos los elementos principales de un proyecto Android. No pierdas de vista este proyecto de ejemplo que hemos creado ya que lo utilizaremos en breve como base para crear nuestra primera aplicación. Pero antes, en el siguiente apartado hablaremos de los componentes software principales con los que podemos construir una aplicación Android.

Componentes de una aplicación Android

En el apartado anterior vimos la estructura de un proyecto Android y aprendimos dónde colocar cada uno de los elementos que componen una aplicación, tanto elementos de software como recursos gráficos o de datos. En éste nuevo post vamos a centrarnos específicamente en los primeros, es decir, veremos los distintos tipos de componentes de software con los que podremos construir una aplicación Android.

En Java o .NET estamos acostumbrados a manejar conceptos como ventana, control, eventos o servicios como los elementos básicos en la construcción de una aplicación.

Pues bien, en Android vamos a disponer de esos mismos elementos básicos aunque con un pequeño cambio en la terminología y el enfoque. Repasemos los componentes principales que pueden formar parte de una aplicación Android [Por claridad, y para evitar confusiones al consultar documentación en inglés, intentaré traducir lo menos posible los nombres originales de los componentes].

Activity

Las actividades (*activities*) representan el componente principal de la interfaz gráfica de una aplicación Android. Se puede pensar en una actividad como el elemento análogo a una ventana o pantalla en cualquier otro lenguaje visual.

View

Las vistas (*view*) son los componentes básicos con los que se construye la interfaz gráfica de la aplicación, análogo por ejemplo a los *controles* de Java o .NET. De inicio, Android pone a nuestra disposición una gran cantidad de controles básicos, como cuadros de texto, botones, listas desplegables o imágenes, aunque también existe la posibilidad de extender la funcionalidad de estos controles básicos o crear nuestros propios controles personalizados.

Service

Los servicios son componentes sin interfaz gráfica que se ejecutan en segundo plano. En concepto, son similares a los servicios presentes en cualquier otro sistema operativo. Los servicios pueden realizar cualquier tipo de acciones, por ejemplo actualizar datos, lanzar notificaciones, o incluso mostrar elementos visuales (p.ej. actividades) si se necesita en algún momento la interacción con del usuario.

Content Provider

Un *content provider* es el mecanismo que se ha definido en Android para compartir datos entre aplicaciones. Mediante estos componentes es posible compartir determinados datos de nuestra aplicación sin mostrar detalles sobre su almacenamiento interno, su estructura, o su implementación. De la misma forma, nuestra aplicación podrá acceder a los datos de otra a través de los *content provider* que se hayan definido.

Broadcast Receiver

Un *broadcast receiver* es un componente destinado a detectar y reaccionar ante determinados mensajes o eventos globales generados por el sistema (por ejemplo: "Batería baja", "SMS recibido", "Tarjeta SD insertada", ...) o por otras aplicaciones (cualquier aplicación puede generar mensajes (*intents*, en terminología Android) broadcast, es decir, no dirigidos a una aplicación concreta sino a cualquiera que quiera escucharlo).

Widget

Los *widgets* son elementos visuales, normalmente interactivos, que pueden mostrarse en la pantalla principal (*home screen*) del dispositivo Android y recibir actualizaciones periódicas. Permiten mostrar información de la aplicación al usuario directamente sobre la pantalla principal.

Intent

Un *intent* es el elemento básico de comunicación entre los distintos componentes Android que hemos descrito anteriormente. Se pueden entender como los mensajes o peticiones que son enviados entre los distintos componentes de una aplicación o entre distintas aplicaciones. Mediante un *intent* se puede mostrar una actividad desde cualquier otra, iniciar un servicio, enviar un mensaje *broadcast*, iniciar otra aplicación, etc.

En el siguiente apartado empezaremos ya a añadir y modificar algo de código, analizando al detalle una aplicación sencilla.

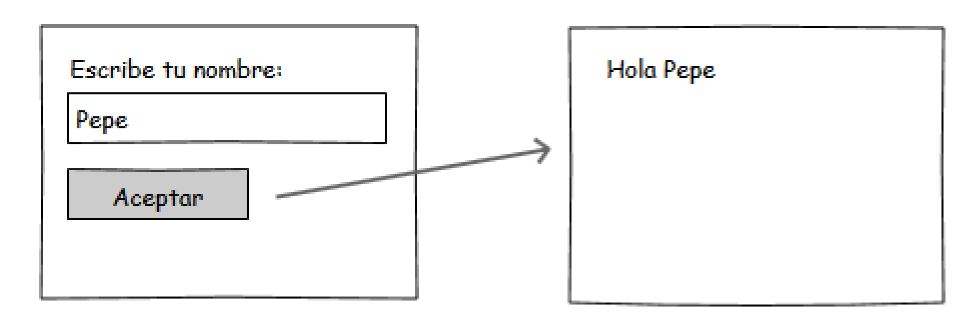
Desarrollando una aplicación Android sencilla

Después de instalar nuestro entorno de desarrollo para Android y comentar la estructura básica de un proyecto y los diferentes componentes software que podemos utilizar ya es hora de empezar a escribir algo de código. Y como siempre lo mejor es empezar por escribir una aplicación sencilla.

En un principio me planteé analizar en este capítulo el clásico *Hola Mundo* pero más tarde me pareció que

se iban a quedar algunas cosas básicas en el tintero. Así que he versionado a mi manera el *Hola Mundo* transformándolo en algo así como un *Hola Usuario*, que es igual de sencilla pero añade un par de cosas interesantes de contar. La aplicación constará de dos pantallas, por un lado la pantalla principal que solicitará un nombre al usuario y una segunda pantalla en la que se mostrará un mensaje personalizado para el usuario. Así de sencillo e inútil, pero aprenderemos muchos conceptos básicos, que para empezar no está mal.

Por dibujarlo para entender mejor lo que queremos conseguir, sería algo tan sencillo como lo siguiente:



Vamos a partir del proyecto de ejemplo que creamos en un apartado anterior, al que casualmente llamamos *HolaUsuario*. Como ya vimos Eclipse había creado por nosotros la estructura de carpetas del proyecto y todos los ficheros necesarios de un *Hola Mundo* básico, es decir, una sola pantalla donde se muestra únicamente un mensaje fijo.

Lo primero que vamos a hacer es diseñar nuestra pantalla principal modificando la que Eclipse nos ha creado por defecto. Aunque ya lo hemos comentado de pasada, recordemos dónde y cómo se define cada pantalla de la aplicación. En Android, el diseño y la lógica de una pantalla están separados en dos ficheros distintos. Por un lado, en el fichero `/res/layout/activity_main.xml` tendremos el diseño puramente visual de la pantalla definido como fichero XML y por otro lado, en el fichero `/src/paquete.java/MainActivity.java`, encontraremos el código java que determina la lógica de la pantalla.

Vamos a modificar en primer lugar el aspecto de la ventana principal de la aplicación añadiendo los controles (*views*) que vemos en el esquema mostrado al principio del apartado. Para ello, vamos a sustituir el contenido del fichero `activity_main.xml` por el siguiente:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/LblNombre"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/nombre" />

    <EditText
        android:id="@+id/TxtNombre"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text" />
```

```
    <Button
        android:id="@+id/BtnHola"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hola" />

</LinearLayout>
```

En este XML se definen los elementos visuales que componen la interfaz de nuestra pantalla principal y se especifican todas sus propiedades. No nos detendremos mucho por ahora en cada detalle, pero expliquemos un poco lo que vemos en el fichero.

Lo primero que nos encontramos es un elemento `LinearLayout`. Los *layout* son elementos no visibles que determinan cómo se van a distribuir en el espacio los controles que incluyamos en su interior. Los programadores java, y más concretamente de *Swing*, conocerán este concepto perfectamente. En este caso, un `LinearLayout` distribuirá los controles simplemente uno tras otro y en la orientación que indique su propiedad `android:orientation`, que en este caso será "vertical".

Dentro del *layout* hemos incluido 3 controles: una etiqueta (`TextView`), un cuadro de texto (`EditText`), y un botón (`Button`). En todos ellos hemos establecido las siguientes propiedades:

- `android:id`. ID del control, con el que podremos identificarlo más tarde en nuestro código. Vemos que el identificador lo escribimos precedido de "`@+id/`". Esto tendrá como efecto que al compilarse el proyecto se genere automáticamente una nueva constante en la clase `R` para dicho control. Así, por ejemplo, como al cuadro de tecto le hemos asignado el ID `TxtNombre`, podremos más tarde acceder al él desde nuestro código haciendo referencia a la constante `R.id.TxtNombre`.

- `android:layout_height` y `android:layout_width`. Dimensiones del control con respecto al layout que lo contiene. Esta propiedad tomará normalmente los valores "`wrap_content`" para indicar que las dimensiones del control se ajustarán al contenido del mismo, o bien "`match_parent`" para indicar que el ancho o el alto del control se ajustará al ancho o alto del layout contenedor respectivamente.

Además de estas propiedades comunes a casi todos los controles que utilizaremos, en el cuadro de texto hemos establecido también la propiedad `android:inputType`, que indica qué tipo de contenido va a albergar el control, en este caso texto normal (valor `"text"`), aunque podría haber sido una contraseña (`textPassword`), un teléfono (`phone`), una fecha (`date`), ….

Por último, en la etiqueta y el botón hemos establecido la propiedad `android:text`, que indica el texto que aparece en el control. Y aquí nos vamos a detener un poco, ya que tenemos dos alternativas a la hora de hacer esto. En Android, el texto de un control se puede especificar directamente como valor de la propiedad `android:text`, o bien utilizar alguna de las cadenas de texto definidas en los recursos del proyecto (como ya vimos, en el fichero `strings.xml`), en cuyo caso indicaremos como valor de la propiedad `android:text` su identificador precedido del prefijo "`@string/`". Dicho de otra forma, la primera alternativa habría sido indicar directamente el texto como valor de la propiedad, por ejemplo en la etiqueta de esta forma:

```
<TextView
    android:id="@+id/LblNombre"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Escribre tu nombre:" />
```

Y la segunda alternativa, la utilizada en el ejemplo, consistiría en definir primero una nueva cadena de texto en el fichero de recursos `/res/values/strings.xml`, por ejemplo con identificador "`nombre`" y valor "`Escribe tu nombre:`"

```
<resources>

    . . .

    <string name="nombre">Escribe tu nombre:</string>

    . . .

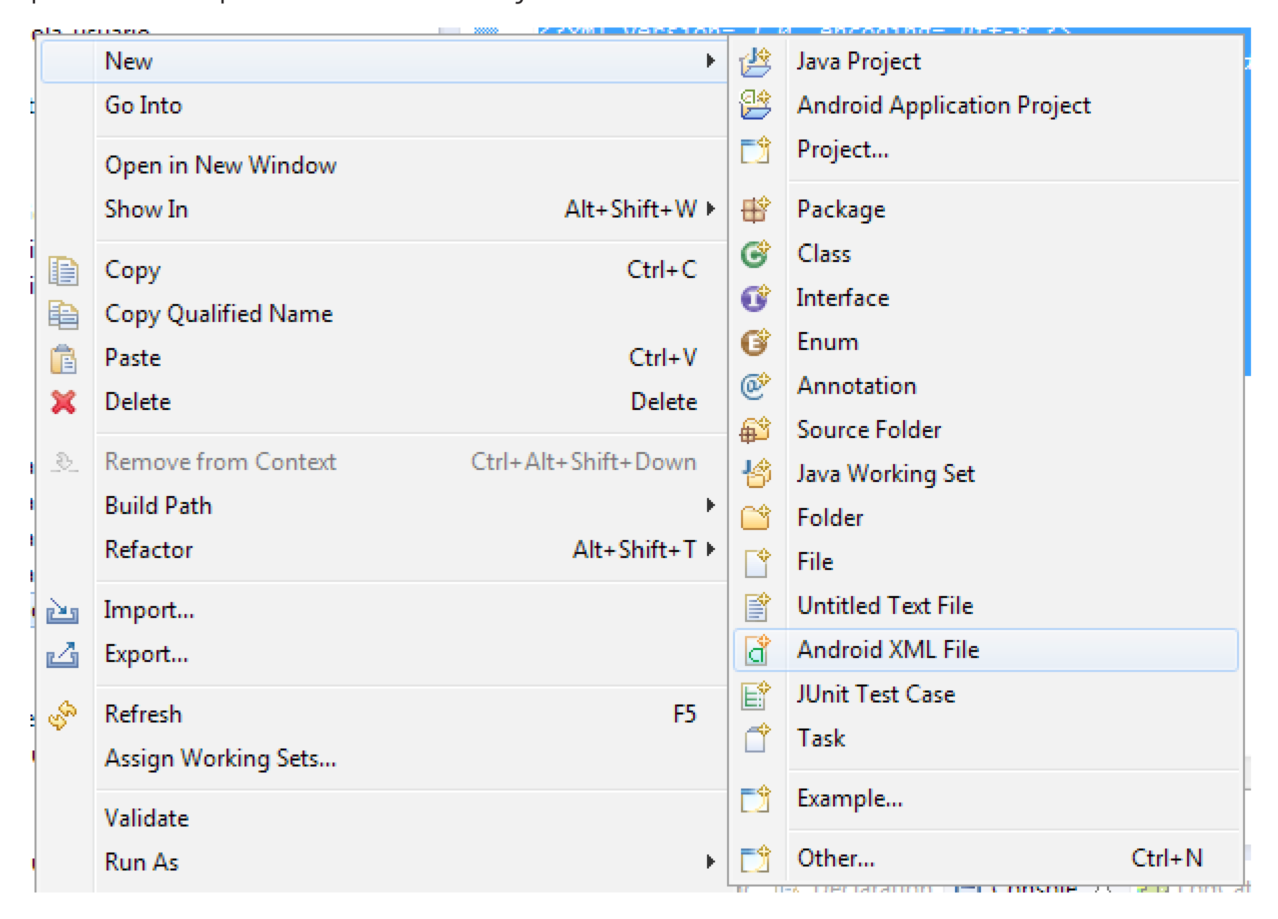
</resources>
```

Y posteriormente indicar el identificador de la cadena como valor de la propiedad `android:text`, siempre precedido del prefijo "`@string/`", de la siguiente forma:

```
<TextView
    android:id="@+id/LblNombre"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/nombre" />
```

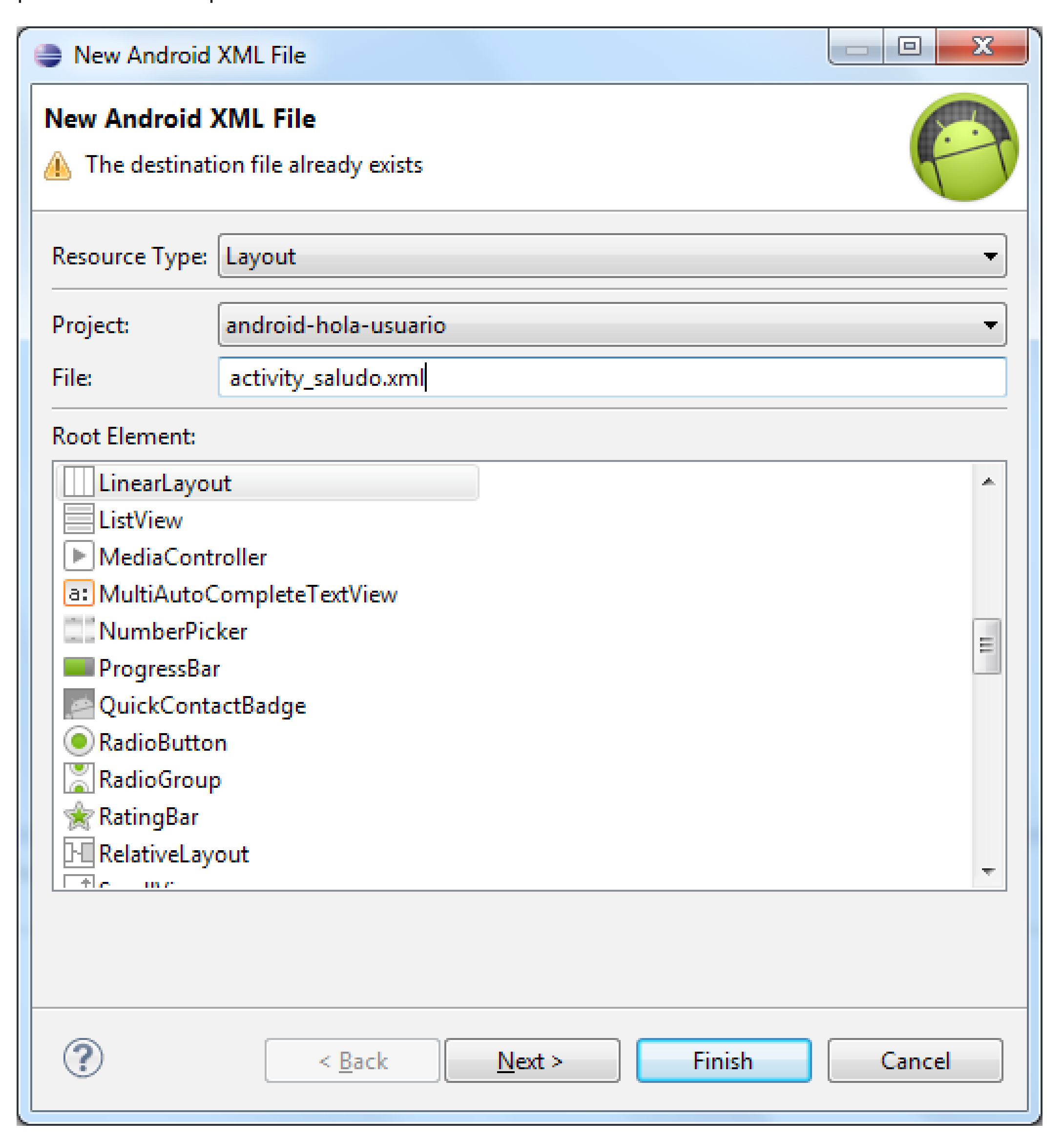
Esta segunda alternativa nos permite tener perfectamente localizadas y agrupadas todas las cadenas de texto utilizadas en la aplicación, lo que nos podría facilitar por ejemplo la traducción de la aplicación a otro idioma.Con esto ya tenemos definida la presentación visual de nuestra ventana principal de la aplicación. De igual forma definiremos la interfaz de la segunda pantalla, creando un nuevo fichero llamado `activity_saludo.xml`, y añadiendo esta vez tan solo una etiqueta (`TextView`) para mostrar el mensaje personalizado al usuario.

Para añadir el fichero, pulsaremos el botón derecho del ratón sobre la carpeta de recursos `/res/layout` y pulsaremos la opción "*New Android XML file*".



En el cuadro de diálogo que nos aparece indicaremos como tipo de recurso "*Layout*", indicaremos el nombre

del fichero (con extensión ".xml") y como elemento raíz seleccionaremos `LinearLayout`. Finalmente pulsamos *Finish* para crear el fichero.



Eclipse creará entonces el nuevo fichero y lo abrirá en el editor gráfico, aunque como ya indicamos, nosotros accederemos a la solapa de código para modificar directamente el contenido XML del fichero.

Para esta segunda pantalla el código que incluiríamos sería el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/TxtSaludo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="" />

</LinearLayout>
```

Una vez definida la interfaz de las pantallas de la aplicación deberemos implementar la lógica de la misma.

Como ya hemos comentado, la lógica de la aplicación se definirá en ficheros java independientes. Para la pantalla principal ya tenemos creado un fichero por defecto llamado `MainActivity.java`. Empecemos por comentar su código por defecto:

```
package net.sgoliver.android.holausuario;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

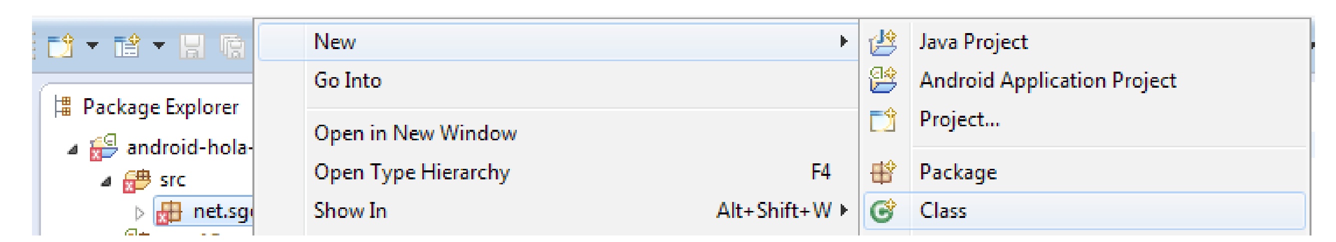
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }
}
```

Como ya vimos en un apartado anterior, las diferentes pantallas de una aplicación Android se definen mediante objetos de tipo `Activity`. Por tanto, lo primero que encontramos en nuestro fichero java es la definición de una nueva clase `MainActivity` que extiende a `Activity`. El único método que modificaremos de esta clase será el método `OnCreate()`, llamado cuando se crea por primera vez la actividad. En este método lo único que encontramos en principio, además de la llamada a su implementación en la clase padre, es la llamada al método `setContentView(R.layout.activity_main)`. Con esta llamada estaremos indicando a Android que debe establecer como interfaz gráfica de esta actividad la definida en el recurso `R.layout.activity_main`, que no es más que la que hemos especificado en el fichero `/res/layout/activity_main.xml`. Una vez más vemos la utilidad de las diferentes constantes de recursos creadas automáticamente en la clase `R` al compilar el proyecto.

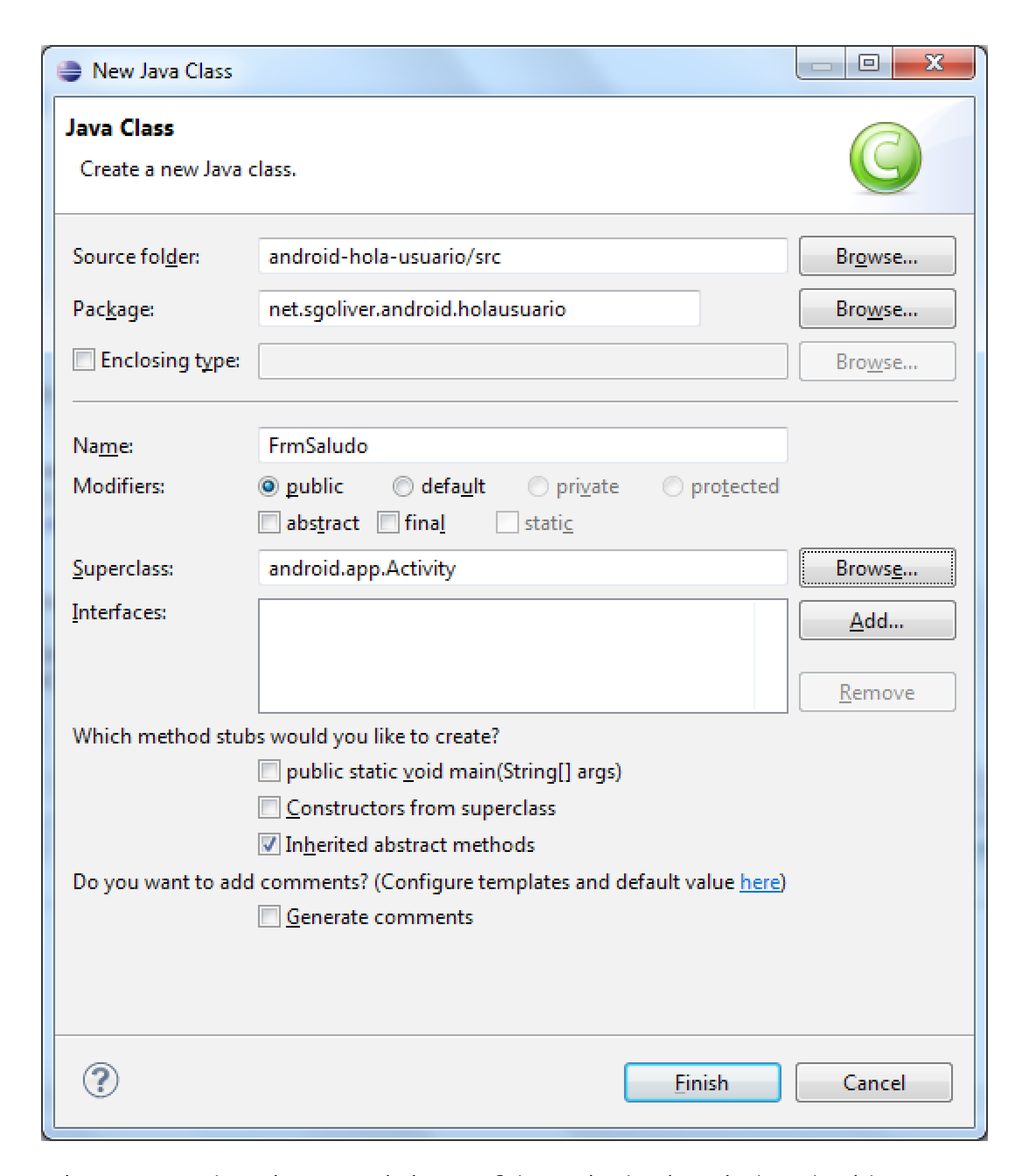
Además del método `onCreate()`, vemos que también se sobrescribe el método `onCreateOptionsMenu()`, que se utiliza para definir menús en la aplicación. Por el momento no tocaremos este método, más adelante en el curso nos ocuparemos de este tema.

Ahora vamos a crear una nueva actividad para la segunda pantalla de la aplicación análoga a ésta primera, para lo que crearemos una nueva clase `FrmSaludo` que extienda también de `Activity` y que implemente el método `onCreate()` pero indicando esta vez que utilice la interfaz definida para la segunda pantalla en `R.layout.activity_saludo`.

Para ello, pulsaremos el botón derecho sobre la carpeta `/src/tu.paquete.java/` y seleccionaremos la opción de menú New / Class.



En el cuadro de diálogo que nos aparece indicaremos el nombre (*Name*) de la nueva clase y su clase padre (*Superclass*) como `android.app.Activity`.



Pulsaremos *Finish* y Eclipse creará el nuevo fichero y lo abrirá en el editor de código java.

Modificaremos por ahora el código de la clase para que quede algo análogo a la actividad principal:

```
package net.sgoliver.android.holausuario;

import android.app.Activity;
import android.os.Bundle;

public class FrmSaludo extends Activity {


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_saludo);
    }
}
```

NOTA: Todos los pasos anteriores de creación de una nueva pantalla (layout xml + clase java) se puede realizar también mediante un asistente de Eclipse al que se accede mediante el menú contextual "New / Other... / Android / Android Activity". Sin embargo, he preferido explicarlo de esta forma para que quedaran claros todos los pasos y elementos necesarios.

Sigamos. Por ahora, el código incluido en estas clases lo único que hace es generar la interfaz de la actividad. A partir de aquí nosotros tendremos que incluir el resto de la lógica de la aplicación.

Y vamos a empezar con la actividad principal `MainActivity`, obteniendo una referencia a los diferentes controles de la interfaz que necesitemos manipular, en nuestro caso sólo el cuadro de texto y el botón. Para ello utilizaremos el método `findViewById()` indicando el ID de cada control, definidos como siempre en la clase `R`. Todo esto lo haremos dentro del método `onCreate()` de la clase `MainActivity`, justo a continuación de la llamada a `setContentView()` que ya comentamos:

```
. . .

import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;

. . .

public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //Obtenemos una referencia a los controles de la interfaz
        final EditText txtNombre = (EditText)findViewById(R.id.TxtNombre);
        final Button btnHola = (Button)findViewById(R.id.BtnHola);

        . . .

    }
}

. . .
```

Como vemos, hemos añadido también varios `import` adicionales para tener acceso a todas las clases utilizadas.

Una vez tenemos acceso a los diferentes controles, ya sólo nos queda implementar las acciones a tomar cuando pulsemos el botón de la pantalla. Para ello, continuando el código anterior, y siempre dentro del método `onCreate()`, implementaremos el evento `onClick` de dicho botón, veamos cómo:

```
. . .

import android.content.Intent;

. . .

public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {

        . . .

        //Obtenemos una referencia a los controles de la interfaz
        final EditText txtNombre = (EditText)findViewById(R.id.TxtNombre);
        final Button btnHola = (Button)findViewById(R.id.BtnHola);

        //Implementamos el evento "click" del botón
        btnHola.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                //Creamos el Intent
                Intent intent =
                        new Intent(MainActivity.this, FrmSaludo.class);

                //Creamos la información a pasar entre actividades
                Bundle b = new Bundle();
                b.putString("NOMBRE", txtNombre.getText().toString());

                //Añadimos la información al intent
                intent.putExtras(b);

                //Iniciamos la nueva actividad
                startActivity(intent);
            }
        });
    }

}

. . .
```

Como ya indicamos en el apartado anterior, la comunicación entre los distintos componentes y aplicaciones en Android se realiza mediante *intents*, por lo que el primer paso será crear un objeto de este tipo. Existen varias variantes del constructor de la clase `Intent`, cada una de ellas dirigida a unas determinadas acciones. En nuestro caso particular vamos a utilizar el intent para llamar a una actividad desde otra actividad de la misma aplicación, para lo que pasaremos a su constructor una referencia a la propia actividad *llamadora* (`MainActivity.this`), y la clase de la actividad *llamada* (`FrmMensaje.class`).

Si quisiéramos tan sólo mostrar una nueva actividad ya tan sólo nos quedaría llamar a `startActivity()` pasándole como parámetro el intent creado. Pero en nuestro ejemplo queremos también pasarle cierta información a la actividad llamada, concretamente el nombre que introduzca el usuario en el cuadro de texto de la pantalla principal. Para hacer esto vamos a crear un objeto `Bundle`, que puede contener una lista de pares *clave-valor* con toda la información a pasar entre las actividades. En nuestro caso sólo añadiremos un dato de tipo `String` mediante el método `putString(clave, valor)`. Tras esto añadiremos la información al intent mediante el método `putExtras(bundle)`.

Con esto hemos finalizado ya actividad principal de la aplicación, por lo que pasaremos ya a la secundaria.

Comenzaremos de forma análoga a la anterior, ampliando el método `onCreate` obteniendo las referencias a los objetos que manipularemos, esta vez sólo la etiqueta de texto. Tras esto viene lo más interesante, debemos recuperar la información pasada desde la actividad principal y asignarla como texto de la etiqueta. Para ello accederemos en primer lugar al intent que ha originado la actividad actual mediante el método `getIntent()` y recuperaremos su información asociada (objeto `Bundle`) mediante el método `getExtras()`.

Hecho esto tan sólo nos queda construir el texto de la etiqueta mediante su método `setText(texto)` y recuperando el valor de nuestra clave almacenada en el objeto `Bundle` mediante `getString(clave)`.

```
package net.sgoliver.android.holausuario;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class FrmSaludo extends Activity {
      @Override
      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_saludo);

            //Localizar los controles
             TextView txtSaludo = (TextView)findViewById(R.id.TxtSaludo);

             //Recuperamos la información pasada en el intent
             Bundle bundle = this.getIntent().getExtras();

             //Construimos el mensaje a mostrar
             txtSaludo.setText("Hola " + bundle.getString("NOMBRE"));
      }
}
```

Con esto hemos concluido la lógica de las dos pantallas de nuestra aplicación y tan sólo nos queda un paso importante para finalizar nuestro desarrollo. Como ya indicamos en un apartado anterior, toda aplicación Android utiliza un fichero especial en formato XML (`AndroidManifest.xml`) para definir, entre otras cosas, los diferentes elementos que la componen. Por tanto, todas las actividades de nuestra aplicación deben quedar convenientemente recogidas en este fichero. La actividad principal ya debe aparecer puesto que se creó de forma automática al crear el nuevo proyecto Android, por lo que debemos añadir tan sólo la segunda.

Para este ejemplo nos limitaremos a incluir la actividad en el XML mediante una nueva etiqueta `<Activity>`, indicar el nombre de la clase java asociada como valor del atributo `android:name`, y asignarle su título mediante el atributo `android:label`, más adelante veremos que opciones adicionales podemos especificar. Todo esto lo incluiremos justo debajo de la definición de la actividad principal dentro del fichero `AndroidManifest.xml`:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="net.sgoliver.android.holausuario"
    android:versionCode="1"
    android:versionName="1.0" >

    . . .

        <activity
            android:name=".MainActivity"
            android:label="@string/title_activity_main" >
```

```
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity android:name=".FrmSaludo"
                  android:label="@string/title_activity_saludo" >

        </activity>

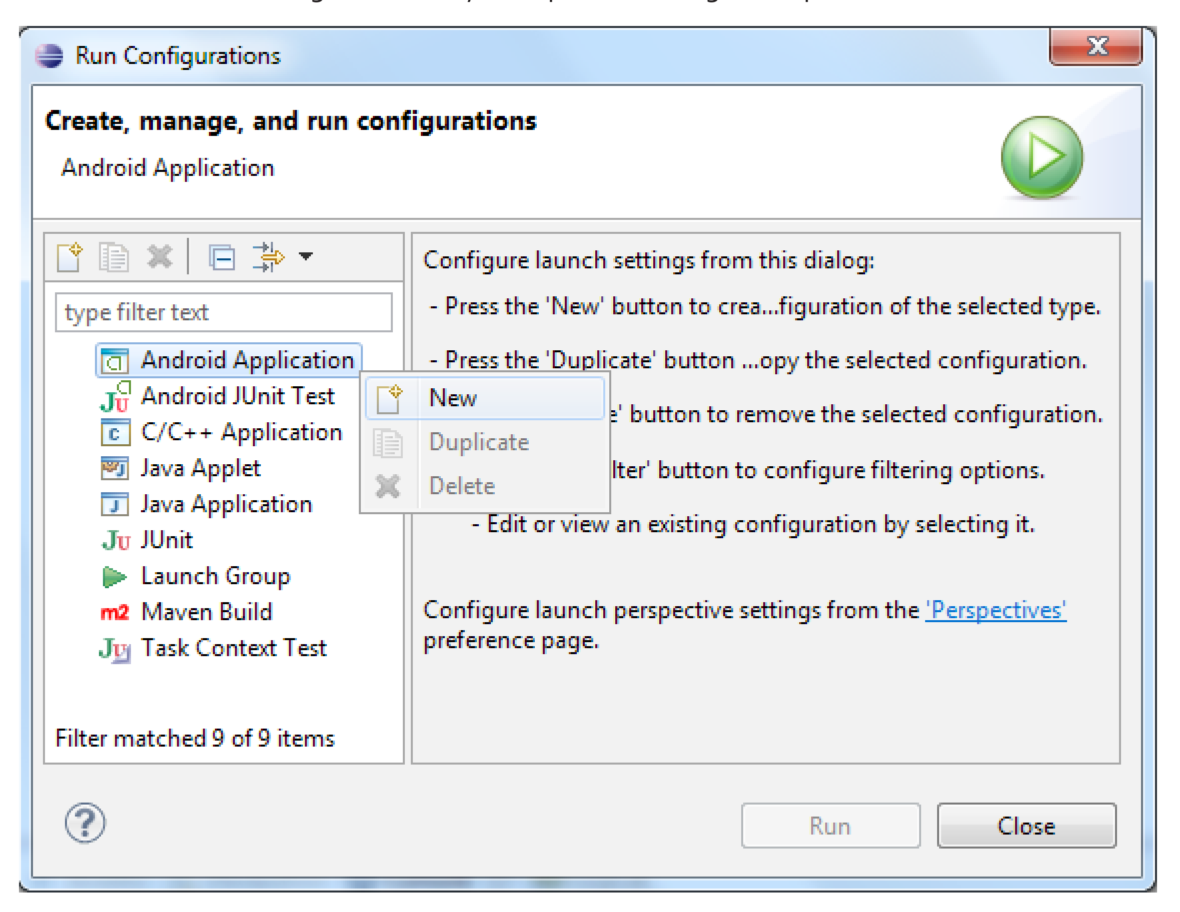
    </application>

</manifest>
```

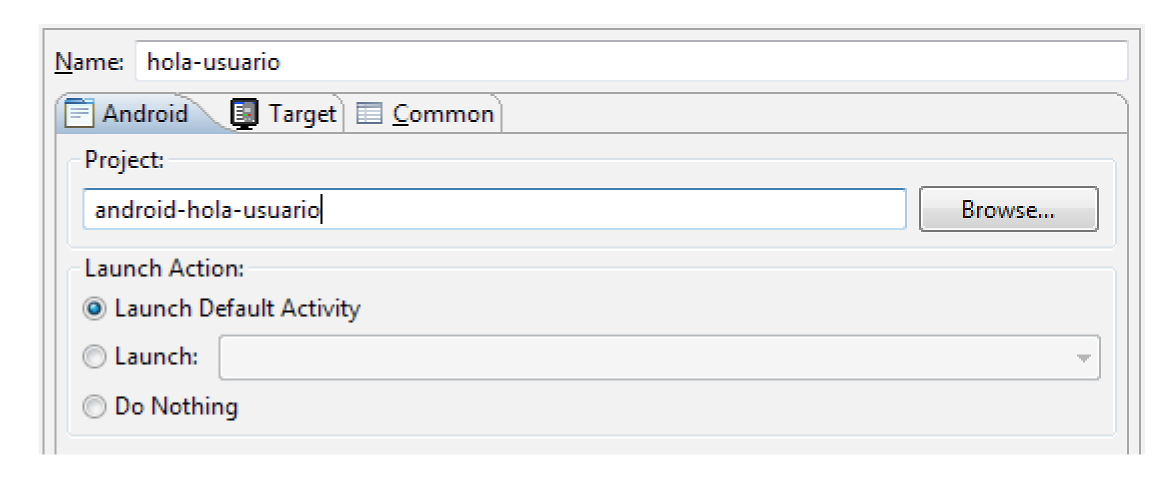
Como vemos, el título de la nueva actividad lo hemos indicado como referencia a una nueva cadena de caracteres, que tendremos que incluir como ya hemos comentado anteriormente en el fichero `/res/values/strings.xml`

Llegados aquí, y si todo ha ido bien, deberíamos poder ejecutar el proyecto sin errores y probar nuestra aplicación en el emulador. La forma de ejecutar y depurar la aplicación en Eclipse es análoga a cualquier otra aplicación java, pero por ser el primer capítulo vamos a recordarla.

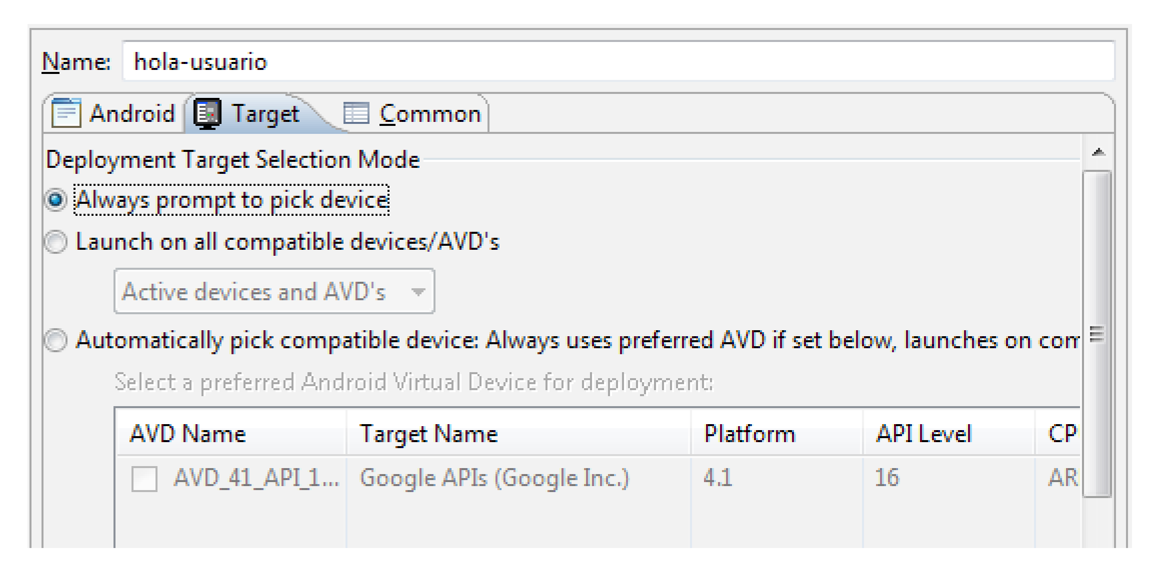
Lo primero que tendremos que hacer será configurar un nuevo "perfil de ejecución". Para ello accederemos al menú "Run/ Run Configurations…" y nos aparecerá la siguiente pantalla.



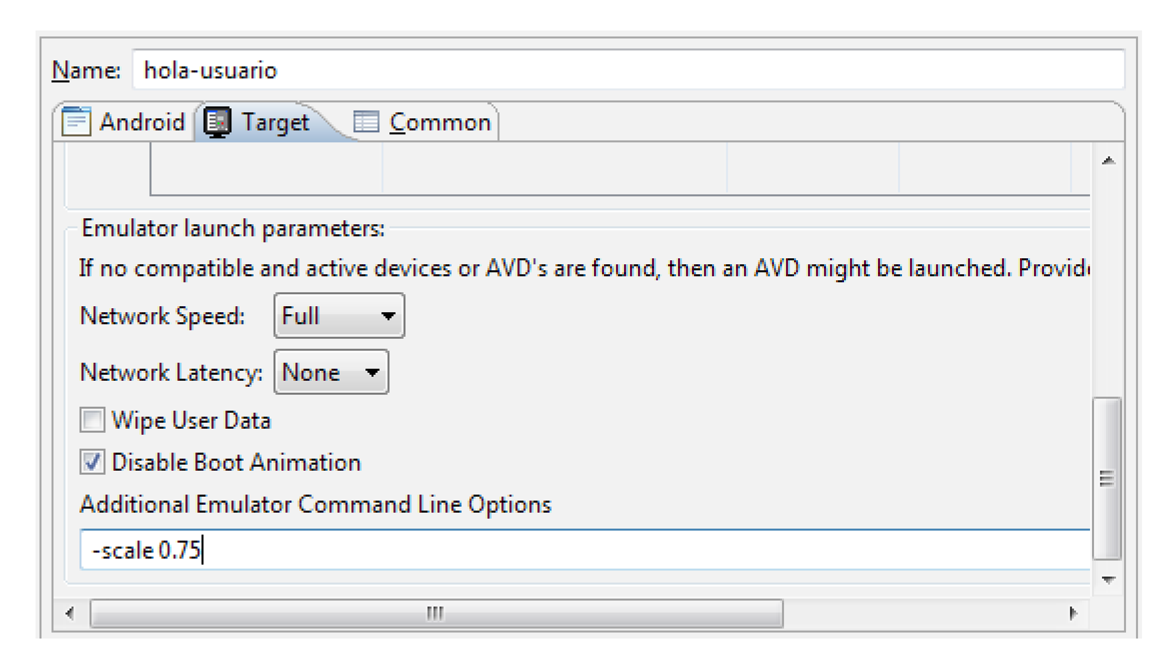
Sobre la categoría "Android Application" pulsaremos el botón derecho y elegiremos la opción "New" para crear un nuevo perfil para nuestra aplicación. En la siguiente pantalla le pondremos un nombre al perfil, en nuestro ejemplo "hola-usuario", y en la pestaña "Android" seleccionaremos el proyecto que queremos ejecutar.



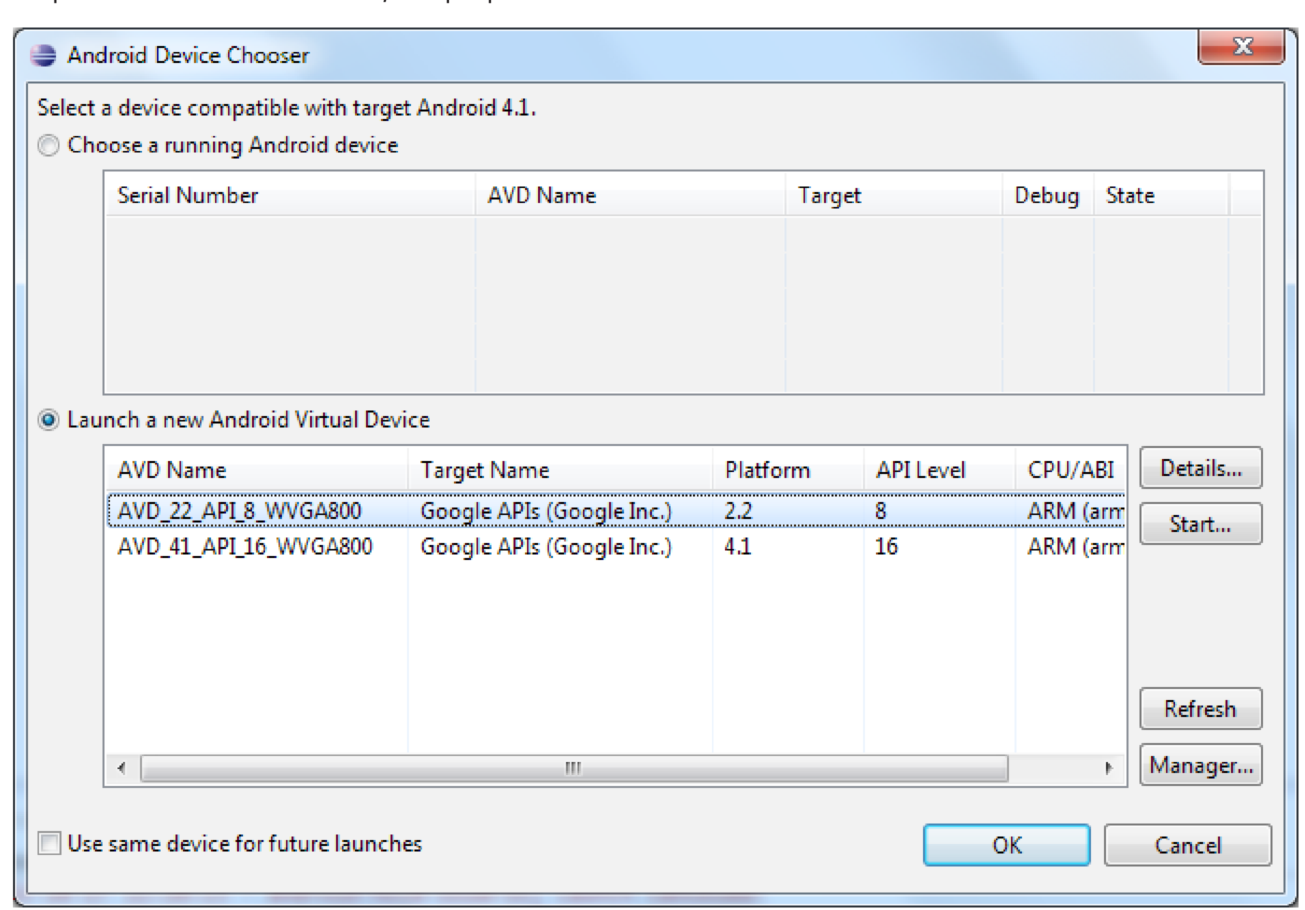
El resto de opciones las dejaremos por defecto y pasaremos a la pestaña "Target". En esta segunda pestaña podremos seleccionar el AVD sobre el que queremos ejecutar la aplicación, aunque suele ser práctico indicarle a Eclipse que nos pregunte esto antes de cada ejecución, de forma que podamos ir alternando fácilmente de AVD sin tener que volver a configurar el perfil. Para ello seleccionaremos la opción "*Always prompt to pick device*".



Un poco más abajo en esta misma pestaña es bueno marcar la opción "*Disable Boot Animation*" para acelerar un poco el primer arranque del emulador, y normalmente también suele ser necesario reducir, o mejor dicho escalar, la pantalla del emulador de forma que podamos verlo completo en la pantalla de nuestro PC. Esto se configura mediante la opción "*Additional Emulator Command Line Options*", donde en mi caso indicaré la opción "`-scale 0.75`", aunque este valor dependerá de la resolución de vuestro monitor y de la configuración del AVD.



Tras esto ya podríamos pulsar el botón "Run" para ejecutar la aplicación en el emulador de Android. Eclipse nos preguntará en qué dispositivo queremos ejecutar y nos mostrará dos listas. La primera de ellas con los dispositivos que haya en ese momento en funcionamiento (por ejemplo si ya teníamos un emulador funcionando) y la siguiente con el resto de AVDs configurados en nuestro entorno. Elegiré en primer lugar el emulador con Android 2.2. Es posible que la primera ejecución se demore unos minutos, todo dependerá de las posibilidades de vuestro PC, así que paciencia.

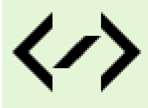


Si todo va bien, tras una pequeña espera aparecerá el emulador de Android y se iniciará automáticamente nuestra aplicación. Podemos probar a escribir un nombre y pulsar el botón "Hola" para comprobar si el funcionamiento es el correcto.



Sin cerrar este emulador podríamos volver a ejecutar la aplicación sobre Android 4.2 seleccionando el AVD correspondiente. De cualquier forma, si vuestro PC no es demasiado potente no recomiendo tener dos emuladores abiertos al mismo tiempo.

Y con esto terminamos por ahora. Espero que esta aplicación de ejemplo os sea de ayuda para aprender temas básicos en el desarrollo para Android, como por ejemplo la definición de la interfaz gráfica, el código java necesario para acceder y manipular los elementos de dicha interfaz, y la forma de comunicar diferentes actividades de Android. En los apartados siguientes veremos algunos de estos temas de forma mucho más específica.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-hola-usuario

2

Interfaz de Usuario

II. Interfaz de Usuario

Layouts

En el apartado anterior, donde desarrollamos una sencilla aplicación Android desde cero, ya hicimos algunos comentarios sobre los *layouts*. Como ya indicamos, los *layouts* son elementos no visuales destinados a controlar la distribución, posición y dimensiones de los controles que se insertan en su interior. Estos componentes extienden a la clase base `ViewGroup`, como muchos otros componentes contenedores, es decir, capaces de contener a otros controles. En el post anterior conocimos la existencia de un tipo concreto de layout, el `LinearLayout`, aunque Android nos proporciona algunos otros. Veamos cuántos y cuáles.

FrameLayout

Éste es el más simple de todos los layouts de Android. Un `FrameLayout` coloca todos sus controles hijos alineados con su esquina superior izquierda, de forma que cada control quedará oculto por el control siguiente (a menos que éste último tenga transparencia). Por ello, suele utilizarse para mostrar un único control en su interior, a modo de contenedor (*placeholder*) sencillo para un sólo elemento sustituible, por ejemplo una imagen.

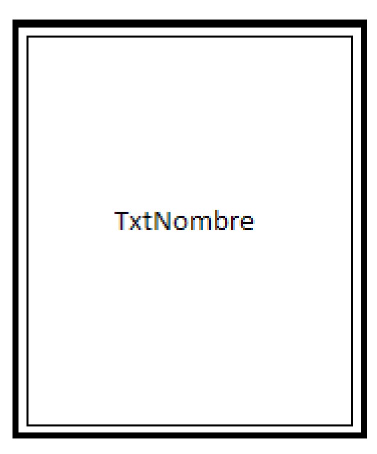
Los componentes incluidos en un `FrameLayout` podrán establecer sus propiedades `android:layout_width` y `android:layout_height`, que podrán tomar los valores "`match_parent`" (para que el control hijo tome la dimensión de su layout contenedor) o "`wrap_content`" (para que el control hijo tome la dimensión de su contenido). **NOTA:** Si estás utilizando una versión de la API de Android inferior a la 8 (Android 2.2), en vez de "`match_parent`" deberás utilizar su equivalente "`fill_parent`".

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <EditText android:id="@+id/TxtNombre"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:inputType="text" />

</FrameLayout>
```

Con el código anterior conseguimos un layout tan sencillo como el siguiente:



LinearLayout

El siguiente tipo de layout en cuanto a nivel de complejidad es el `LinearLayout`. Este layout apila

uno tras otro todos sus elementos hijos de forma horizontal o vertical según se establezca su propiedad `android:orientation`.

Al igual que en un `FrameLayout`, los elementos contenidos en un `LinearLayout` pueden establecer sus propiedades `android:layout_width` y `android:layout_height` para determinar sus dimensiones dentro del layout. Pero en el caso de un `LinearLayout`, tendremos otro parámetro con el que jugar, la propiedad `android:layout_weight`.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <EditText android:id="@+id/TxtNombre"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <Button android:id="@+id/BtnAceptar"
        android:layout_width="wrap_content"
        android:layout_height="match_parent" />

</LinearLayout>
```

Esta propiedad nos va a permitir dar a los elementos contenidos en el layout unas dimensiones proporcionales entre ellas. Esto es más difícil de explicar que de comprender con un ejemplo. Si incluimos en un `LinearLayout` vertical dos cuadros de texto (`EditText`) y a uno de ellos le establecemos un `layout_weight="1"` y al otro un `layout_weight="2"` conseguiremos como efecto que toda la superficie del layout quede ocupada por los dos cuadros de texto y que además el segundo sea el doble (relación entre sus propiedades `weight`) de alto que el primero.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <EditText android:id="@+id/TxtDato1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:inputType="text"
        android:layout_weight="1" />

    <EditText android:id="@+id/TxtDato2"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:inputType="text"
        android:layout_weight="2" />

</LinearLayout>
```

Con el código anterior conseguiríamos un layout como el siguiente:



Así pues, a pesar de la simplicidad aparente de este layout resulta ser lo suficiente versátil como para sernos de utilidad en muchas ocasiones.

TableLayout

Un `TableLayout` permite distribuir sus elementos hijos de forma tabular, definiendo las filas y columnas necesarias, y la posición de cada componente dentro de la tabla.

La estructura de la tabla se define de forma similar a como se hace en HTML, es decir, indicando las filas que compondrán la tabla (objetos `TableRow`), y dentro de cada fila las columnas necesarias, con la salvedad de que no existe ningún objeto especial para definir una columna (algo así como un *`TableColumn`*) sino que directamente insertaremos los controles necesarios dentro del `TableRow` y cada componente insertado (que puede ser un control sencillo o incluso otro `ViewGroup`) corresponderá a una columna de la tabla. De esta forma, el número final de filas de la tabla se corresponderá con el número de elementos `TableRow` insertados, y el número total de columnas quedará determinado por el número de componentes de la fila que más componentes contenga.

Por norma general, el ancho de cada columna se corresponderá con el ancho del mayor componente de dicha columna, pero existen una serie de propiedades que nos ayudarán a modificar este comportamiento:

- `android:stretchColumns`. Indicará las columnas que pueden expandir para absorver el espacio libre dejado por las demás columnas a la derecha de la pantalla.
- `android:shrinkColumns`. Indicará las columnas que se pueden contraer para dejar espacio al resto de columnas que se puedan salir por la derecha de la pantalla.
- `android:collapseColumns`. Indicará las columnas de la tabla que se quieren ocultar completamente.

Todas estas propiedades del `TableLayout` pueden recibir una lista de índices de columnas separados por comas (ejemplo: `android:stretchColumns="1,2,3"`) o un asterisco para indicar que debe aplicar a todas las columnas (ejemplo: `android:stretchColumns="*"`).

Otra característica importante es la posibilidad de que una celda determinada pueda ocupar el espacio de varias columnas de la tabla (análogo al atributo `colspan` de HTML). Esto se indicará mediante la propiedad `android:layout_span` del componente concreto que deberá tomar dicho espacio.

Veamos un ejemplo con varios de estos elementos:

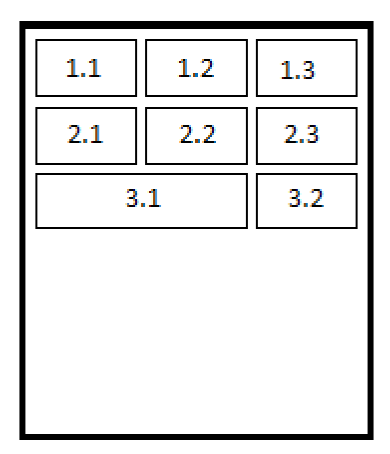
```
<TableLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
```

```
    <TableRow>
        <TextView android:text="Celda 1.1" />
        <TextView android:text="Celda 1.2" />
        <TextView android:text="Celda 1.3" />
    </TableRow>

    <TableRow>
        <TextView android:text="Celda 2.1" />
        <TextView android:text="Celda 2.2" />
        <TextView android:text="Celda 2.3" />
    </TableRow>

    <TableRow>
        <TextView android:text="Celda 3.1"
               android:layout_span="2" />
        <TextView android:text="Celda 3.2" />
    </TableRow>
</TableLayout>
```

El layout resultante del código anterior sería el siguiente:



GridLayout

Este tipo de layout fue incluido a partir de la API 14 (Android 4.0) y sus características son similares al `TableLayout`, ya que se utiliza igualmente para distribuir los diferentes elementos de la interfaz de forma tabular, distribuidos en filas y columnas. La diferencia entre ellos estriba en la forma que tiene el `GridLayout` de colocar y distribuir sus elementos hijos en el espacio disponible. En este caso, a diferencia del `TableLayout` indicaremos el número de filas y columnas como propiedades del layout, mediante `android:rowCount` y `android:columnCount`. Con estos datos ya no es necesario ningún tipo de elemento para indicar las filas, como el elemento `TableRow` del `TableLayout`, sino que los diferentes elementos hijos se irán colocando ordenadamente por filas o columnas (dependiendo de la propiedad `android:orientation`) hasta completar el número de filas o columnas indicadas en los atributos anteriores. Adicionalmente, igual que en el caso anterior, también tendremos disponibles las propiedades `android:layout_rowSpan` y `android:layout_columnSpan` para conseguir que una celda ocupe el lugar de varias filas o columnas.

Existe también una forma de indicar de forma explícita la fila y columna que debe ocupar un determinado elemento hijo contenido en el `GridLayout`, y se consigue utilizando los atributos `android:layout_row` y `android:layout_column`. De cualquier forma, salvo para configuraciones complejas del grid no suele ser necesario utilizar estas propiedades.

Con todo esto en cuenta, para conseguir una distribución equivalente a la del ejemplo anterior del `TableLayout`, necesitaríamos escribir un código como el siguiente:

```
<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:rowCount="2"
    android:columnCount="3"
    android:orientation="horizontal" >

        <TextView android:text="Celda 1.1" />
        <TextView android:text="Celda 1.2" />
        <TextView android:text="Celda 1.3" />

        <TextView android:text="Celda 2.1" />
        <TextView android:text="Celda 2.2" />
        <TextView android:text="Celda 2.3" />

        <TextView android:text="Celda 3.1"
               android:layout_columnSpan="2" />

        <TextView android:text="Celda 3.2" />

</GridLayout>
```

RelativeLayout

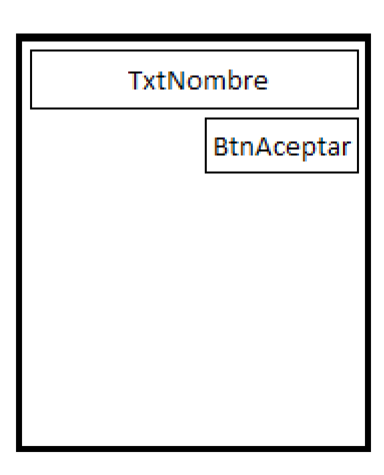
El último tipo de layout que vamos a ver es el `RelativeLayout`. Este layout permite especificar la posición de cada elemento de forma relativa a su elemento padre o a cualquier otro elemento incluido en el propio layout. De esta forma, al incluir un nuevo elemento A podremos indicar por ejemplo que debe colocarse debajo del elemento B y alineado a la derecha del layout padre. Veamos esto en el ejemplo siguiente:

```
<RelativeLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

        <EditText android:id="@+id/TxtNombre"
              android:layout_width="match_parent"
              android:layout_height="wrap_content"
              android:inputType="text" />

        <Button android:id="@+id/BtnAceptar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_below="@id/TxtNombre"
            android:layout_alignParentRight="true" />
</RelativeLayout>
```

En el ejemplo, el botón `BtnAceptar` se colocará debajo del cuadro de texto `TxtNombre` (`android:layout_below="@id/TxtNombre"`) y alineado a la derecha del layout padre (`android:layout_alignParentRight="true"`). Quedaría algo así:



Al igual que estas tres propiedades, en un `RelativeLayout` tendremos un sinfín de propiedades para colocar cada control justo donde queramos. Veamos las principales [creo que sus propios nombres explican perfectamente la función de cada una]:

Tipo	Propiedades

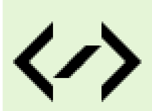
Posición relativa a otro control	`android:layout_above`
	`android:layout_below`
	`android:layout_toLeftOf`
	`android:layout_toRightOf`
	`android:layout_alignLeft`
	`android:layout_alignRight`
	`android:layout_alignTop`
	`android:layout_alignBottom`
	`android:layout_alignBaseline`

Posición relativa al layout padre	`android:layout_alignParentLeft`
	`android:layout_alignParentRight`
	`android:layout_alignParentTop`
	`android:layout_alignParentBottom`
	`android:layout_centerHorizontal`
	`android:layout_centerVertical`
	`android:layout_centerInParent`

Opciones de margen (también disponibles para el resto de layouts)	`android:layout_margin`
	`android:layout_marginBottom`
	`android:layout_marginTop`
	`android:layout_marginLeft`
	`android:layout_marginRight`

Opciones de espaciado o *padding* (también disponibles para el resto de layouts)	android:padding android:paddingBottom android:paddingTop android:paddingLeft android:paddingRight

En próximos apartados veremos otros elementos comunes que extienden a ViewGroup, como por ejemplo las vistas de tipo lista (ListView), de tipo grid (GridView), y en pestañas (TabHost/TabWidget).



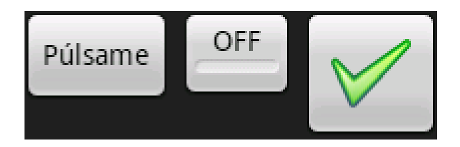
Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-layouts

Botones

En el apartado anterior hemos visto los distintos tipos de *layout* con los que contamos en Android para distribuir los controles de la interfaz por la pantalla del dispositivo. En los próximos apartados vamos a hacer un repaso de los diferentes controles que pone a nuestra disposición la plataforma de desarrollo de este sistema operativo. Empezaremos con los controles más básicos y seguiremos posteriormente con algunos algo más elaborados.

En esta primera parada sobre el tema nos vamos a centrar en los diferentes tipos de botones y cómo podemos personalizarlos. El SDK de Android nos proporciona tres tipos de botones: el clásico (Button), el de tipo *on/off* (ToggleButton), y el que puede contener una imagen (ImageButton). En la imagen siguiente vemos el aspecto por defecto de estos tres controles.



No vamos a comentar mucho sobre ellos dado que son controles de sobra conocidos por todos, ni vamos a enumerar todas sus propiedades porque existen decenas. A modo de referencia, a medida que los vayamos comentando iré poniendo enlaces a su página de la documentación oficial de Android para poder consultar todas sus propiedades en caso de necesidad.

Control Button [API]

Un control de tipo Button es el botón más básico que podemos utilizar. En el ejemplo siguiente definimos un botón con el texto "*Púlsame*" asignando su propiedad android:text. Además de esta propiedad podríamos utilizar muchas otras como el color de fondo (android:background), estilo de fuente (android:typeface), color de fuente (android:textcolor), tamaño de fuente (android:textSize), etc.

```
<Button
      android:id="@+id/BtnBoton1"
      android:text="@string/pulsame"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
```

Control ToggleButton [API]

Un control de tipo `ToggleButton` es un tipo de botón que puede permanecer en dos posibles estados, pulsado/no_pulsado. En este caso, en vez de definir un sólo texto para el control definiremos dos, dependiendo de su estado. Así, podremos asignar las propiedades `android:textOn` y `android:textoOff` para definir ambos textos.

Veamos un ejemplo a continuación:

```
<ToggleButton android:id="@+id/BtnBoton2"
      android:textOn="@string/on"
      android:textOff="@string/off"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
```

Control ImageButton [API]

En un control de tipo `ImageButton` podremos definir una imagen a mostrar en vez de un texto, para lo que deberemos asignar la propiedad `android:src`. Normalmente asignaremos esta propiedad con el descriptor de algún recurso que hayamos incluido en la carpeta `/res/drawable`. Así, por ejemplo, en nuestro caso hemos incluido una imagen llamada "*ok.png*" por lo que haremos referencia al recurso "`@drawable/ok`". Adicionalmente, al tratarse de un control de tipo imagen también deberíamos acostumbrarnos a asignar la propiedad `android:contentDescription` con una descripción textual de la imagen, de forma que nuestra aplicación sea lo más *accesible* posible.

```
<ImageButton android:id="@+id/BtnBoton3"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:contentDescription="@string/icono_ok"
      android:src="@drawable/ok" />
```

Cabe decir además, que aunque existe este tipo específico de botón para imágenes, también es posible añadir una imagen a un botón normal de tipo `Button`, a modo de elemento suplementario al texto. Por ejemplo, si quisiéramos añadir un icono a la izquierda del texto de un botón utilizaríamos la propiedad `android:drawableLeft` indicando como valor el descriptor (ID) de la imagen que queremos mostrar:

```
<Button android:id="@+id/BtnBoton5"
      android:text="@string/pulsame"
      android:drawableLeft="@drawable/ok"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
```

El botón mostrado en este caso sería similar a éste:



Eventos de un botón

Como podéis imaginar, aunque estos controles pueden lanzar muchos otros eventos, el más común de todos ellos y el que querremos capturar en la mayoría de las ocasiones es el evento `onClick`, que se lanza cada vez que el usuario pulsa el botón. Para definir la lógica de este evento tendremos que implementarla definiendo un nuevo objeto `View.OnClickListener()` y asociándolo al botón mediante el método `setOnClickListener()`. La forma más habitual de hacer esto es la siguiente:

```
private Button btnBoton1;

//...

btnBoton1 = (Button)findViewById(R.id.BtnBoton1);

btnBoton1.setOnClickListener(new View.OnClickListener() {
	public void onClick(View arg0)
	{
		lblMensaje.setText("Botón 1 pulsado!");
	}
});
```

En el caso de un botón de tipo `ToggleButton` suele ser de utilidad conocer en qué estado ha quedado el botón tras ser pulsado, para lo que podemos utilizar su método `isChecked()`. En el siguiente ejemplo se comprueba el estado del botón tras ser pulsado y se realizan acciones distintas según el resultado.

```
btnBoton2 = (ToggleButton)findViewById(R.id.BtnBoton2);

btnBoton2.setOnClickListener(new View.OnClickListener() {
	public void onClick(View arg0)
	{
		if(btnBoton2.isChecked())
			lblMensaje.setText("Botón 2: ON");
		else
			lblMensaje.setText("Botón 2: OFF");
	}
});
```

Personalizar el aspecto un botón [y otros controles]

En la imagen mostrada al principio del apartado vimos el aspecto que presentan por defecto los tres tipos de botones disponibles. Pero, ¿y si quisiéramos personalizar su aspecto más allá de cambiar un poco el tipo o el color de la letra o el fondo?

Para cambiar la forma de un botón podríamos simplemente asignar una imagen a la propiedad `android:background`, pero esta solución no nos serviría de mucho porque siempre se mostraría la misma imagen incluso con el botón pulsado, dando poca sensación de elemento "*clickable*".

La solución perfecta pasaría por tanto por definir diferentes imágenes de fondo dependiendo del estado del botón. Pues bien, Android nos da total libertad para hacer esto mediante el uso de *selectores*. Un selector se define mediante un fichero XML localizado en la carpeta `/res/drawable`, y en él se pueden establecer los diferentes valores de una propiedad determinada de un control dependiendo de su estado.

Por ejemplo, si quisiéramos dar un aspecto plano a un botón `ToggleButton`, podríamos diseñar las imágenes necesarias para los estados "pulsado" (en el ejemplo `toggle_on.png`) y "no pulsado" (en el ejemplo `toggle_off.png`) y crear un selector como el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<selector
    xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:state_checked="false"
            android:drawable="@drawable/toggle_off" />
    <item android:state_checked="true"
            android:drawable="@drawable/toggle_on" />
</selector>
```

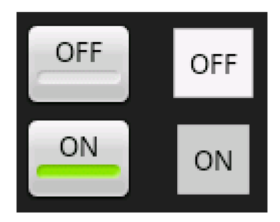
En el código anterior vemos cómo se asigna a cada posible estado del botón una imagen (un elemento *drawable*) determinada. Así, por ejemplo, para el estado "pulsado" (`state_checked="true"`) se asigna la imagen `toggle_on`.

Este selector lo guardamos por ejemplo en un fichero llamado `toggle_style.xml` y lo colocamos como un recurso más en nuestra carpeta de recursos `/res/drawable`.

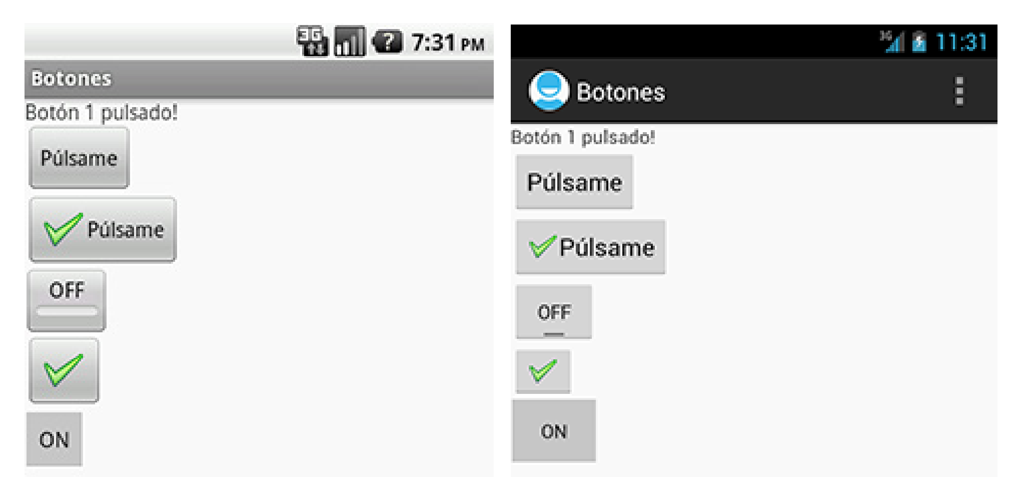
Hecho esto, tan sólo bastaría hacer referencia a este nuevo recurso que hemos creado en la propiedad `android:background` del botón:

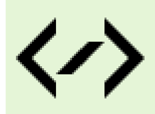
```
<ToggleButton
        android:id="@+id/BtnBoton4"
        android:textOn="@string/on"
        android:textOff="@string/off"
        android:padding="10dip"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="@drawable/toggle_style"/>
```

En la siguiente imagen vemos el aspecto por defecto de un `ToggleButton` (columna izquierda) y cómo ha quedado nuestro `ToggleButton` personalizado (columna derecha).



En las imágenes siguientes se muestra la aplicación de ejemplo desarrollada, donde se puede comprobar el aspecto de cada uno de los tipos de botón comentados para las versiones de Android 2.x y 4.x





Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-botones

Imágenes, etiquetas y cuadros de texto

En este apartado nos vamos a centrar en otros tres componentes básicos imprescindibles en nuestras aplicaciones: las imágenes (`ImageView`), las etiquetas (`TextView`) y por último los cuadros de texto

(EditText).

Control ImageView [API]

El control ImageView permite mostrar imágenes en la aplicación. La propiedad más interesante es android:src, que permite indicar la imagen a mostrar. Nuevamente, lo normal será indicar como origen de la imagen el identificador de un recurso de nuestra carpeta /res/drawable, por ejemplo android:src="@drawable/unaimagen". Además de esta propiedad, existen algunas otras útiles en algunas ocasiones como las destinadas a establecer el tamaño máximo que puede ocupar la imagen, android:maxWidth y android:maxHeight, o para indicar cómo debe adaptarse la imagen al tamaño del control, android:scaleType (5=CENTER, 6=CENTER_CROP, 7=CENTER_INSIDE, …). Además, como ya comentamos para el caso de los controles ImageButton, al tratarse de un control de tipo imagen deberíamos establecer siempre la propiedad android:contentDescription para ofrecer una breve descripción textual de la imagen, algo que hará nuestra aplicación mucho más accesible.

```
<ImageView android:id="@+id/ImgFoto"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:src="@drawable/icon"
android:contentDescription="@string/imagen_ejemplo" />
```

Si en vez de establecer la imagen a mostrar en el propio layout XML de la actividad quisiéramos establecerla mediante código utilizaríamos el método setImageResorce(…), pasándole el ID del recurso a utilizar como contenido de la imagen.

```
ImageView img = (ImageView)findViewById(R.id.ImgFoto);
img.setImageResource(R.drawable.icon);
```

En cuanto a posibles eventos, al igual que comentamos para los controles de tipo botón en el apartado anterior, para los componentes ImageView también podríamos implementar su evento onClick, de forma idéntica a la que ya vimos, aunque en estos casos suele ser menos frecuente la necesidad de capturar este evento.

Control TextView [API]

El control TextView es otro de los clásicos en la programación de GUIs, las etiquetas de texto, y se utiliza para mostrar un determinado texto al usuario. Al igual que en el caso de los botones, el texto del control se establece mediante la propiedad android:text. A parte de esta propiedad, la naturaleza del control hace que las más interesantes sean las que establecen el formato del texto mostrado, que al igual que en el caso de los botones son las siguientes: android:background (color de fondo), android:textColor (color del texto), android:textSize (tamaño de la fuente) y android:typeface (estilo del texto: negrita, cursiva, …).

```
<TextView android:id="@+id/LblEtiqueta"
     android:layout_width="match_parent"
     android:layout_height="wrap_content"
     android:text="@string/escribe_algo"
     android:background="#AA44FF"
     android:typeface="monospace" />
```

De igual forma, también podemos manipular estas propiedades desde nuestro código. Como ejemplo, en el siguiente fragmento recuperamos el texto de una etiqueta con getText(), y posteriormente le concatenamos unos números, actualizamos su contenido mediante setText() y le cambiamos su color de fondo con setBackgroundColor().

```
final TextView lblEtiqueta = (TextView)findViewById(R.id.LblEtiqueta);
String texto = lblEtiqueta.getText().toString();
texto += "123";
lblEtiqueta.setText(texto);
lblEtiqueta.setBackgroundColor(Color.BLUE);
```

Control EditText [API]

El control `EditText` es el componente de edición de texto que proporciona la plataforma Android. Permite la introducción y edición de texto por parte del usuario, por lo que en tiempo de diseño la propiedad más interesante a establecer, además de su posición, tamaño y formato, es el texto a mostrar, atributo `android:text`. Por supuesto si no queremos que el cuadro de texto aparezca inicializado con ningún texto, no es necesario incluir esta propiedad en el layout XML. Lo que sí deberemos establecer será la propiedad `android:inputType`. Esta propiedad indica el tipo de contenido que se va a introducir en el cuadro de texto, como por ejemplo una dirección de correo electrónico (`textEmailAddress`), un número genérico (`number`), un número de teléfono (`phone`), una dirección web (`textUri`), o un texto genérico (`text`). El valor que establezcamos para esta propiedad tendrá además efecto en el tipo de teclado que mostrará Android para editar dicho campo. Así, por ejemplo, si hemos indicado "`text`" mostrará el teclado completo alfanumérico, si hemos indicado "`phone`" mostrará el teclado numérico del teléfono, etc.

```
<EditText android:id="@+id/TxtTexto"
     android:layout_width="match_parent"
     android:layout_height="wrap_content"
     android:inputType="text" />
```

De igual forma, desde nuestro código podremos recuperar y establecer este texto mediante los métodos `getText()` y `setText(nuevoTexto)` respectivamente:

```
final EditText txtTexto = (EditText)findViewById(R.id.TxtTexto);
String texto = txtTexto.getText().toString();
txtTexto.setText("Hola mundo!");
```

Un detalle que puede haber pasado desapercibido. ¿Os habéis fijado en que hemos tenido que hacer un `toString()` sobre el resultado de `getText()`? La explicación para esto es que el método `getText()` no devuelve directamente una cadena de caracteres (`String`) sino un objeto de tipo `Editable`, que a su vez implementa la interfaz `Spannable`. Y esto nos lleva a la característica más interesante del control `EditText`, y es que no sólo nos permite editar texto plano sino también texto enriquecido o con formato. Veamos cómo y qué opciones tenemos disponibles, y para empezar comentemos algunas cosas sobre los objetos `Spannable`.

Interfaz Spanned

Un objeto de tipo `Spanned` es algo así como una cadena de caracteres (de hecho deriva de la interfaz `CharSequence`) en la que podemos insertar otros objetos a modo de marcas o etiquetas (*spans*) asociados a rangos de caracteres. De esta interfaz deriva la interfaz `Spannable`, que permite la modificación de estas marcas, y a su vez de ésta última deriva la interfaz `Editable`, que permite además la modificación del texto.

Aunque en el apartado en el que nos encontramos nos interesaremos principalmente por las marcas de formato de texto, en principio podríamos insertar cualquier tipo de objeto.

Existen muchos tipos de *spans* predefinidos en la plataforma que podemos utilizar para dar formato al texto, entre ellos:

- `TypefaceSpan`. Modifica el tipo de fuente.
- `StyleSpan`. Modifica el estilo del texto (negrita, cursiva, …).

- ForegroudColorSpan. Modifica el color del texto.
- AbsoluteSizeSpan. Modifica el tamaño de fuente.

De esta forma, para crear un nuevo objeto Editable e insertar una marca de formato podríamos hacer lo siguiente:

```
//Creamos un nuevo objeto de tipo Editable
Editable str = Editable.Factory.getInstance().newEditable("Esto es un
simulacro.");

//Marcamos cono fuente negrita la palabra "simulacro" (caracteres del 11-19)
str.setSpan(new StyleSpan(android.graphics.Typeface.BOLD), 11, 19, Spannable.
SPAN_EXCLUSIVE_EXCLUSIVE);
```

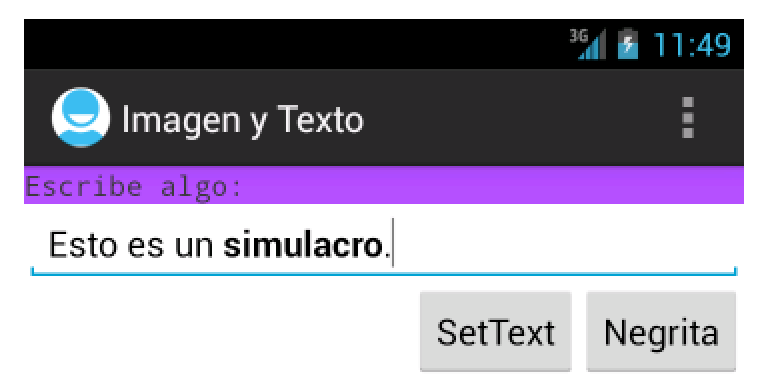
En este ejemplo estamos insertando un span de tipo StyleSpan para marcar un fragmento de texto con estilo negrita. Para insertarlo utilizamos el método setSpan(), que recibe como parámetro el objeto Span a insertar, la posición inicial y final del texto a marcar, y un *flag* que indica la forma en la que el span se podrá extender al insertarse nuevo texto.

Texto con formato en controles TextView y EditText

Hemos visto cómo crear un objeto Editable y añadir marcas de formato al texto que contiene, pero todo esto no tendría ningún sentido si no pudiéramos visualizarlo. Como ya podéis imaginar, los controles TextView y EditText nos van a permitir hacer esto. Vamos qué ocurre si asignamos a nuestro control EditText el objeto Editable que hemos creado antes:

```
txtTexto.setText(str);
```

Tras ejecutar este código, para lo que hemos insertado un botón "SetText" en la aplicación de ejemplo, veremos cómo efectivamente en el cuadro de texto aparece el mensaje con el formato esperado:



Como podéis ver en la captura anterior, en la aplicación de ejemplo también he incluido un botón adicional "Negrita" que se encargará de convertir a estilo negrita un fragmento de texto previamente seleccionado en el cuadro de texto. Mi intención con esto es presentar los métodos disponibles para determinar el comienzo y el fin de una selección en un control de este tipo. Para ello utilizaremos los métodos getSelectionStart() y getSelectionEnd(), que nos devolverán el índice del primer y último carácter seleccionado en el texto. Sabiendo esto, ya sólo nos queda utilizar el método setSpan() que ya conocemos para convertir la selección a negrita.

```
Spannable texto = txtTexto.getText();

int ini = txtTexto.getSelectionStart();
int fin = txtTexto.getSelectionEnd();

texto.setSpan(
      new StyleSpan(android.graphics.Typeface.BOLD),
      ini, fin,
      Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);
```

Bien, ya hemos visto cómo asignar texto con y sin formato a un cuadro de texto, pero ¿qué ocurre a la hora de recuperar texto con formato desde el control?. Ya vimos que la función `getText()` devuelve un objeto de tipo `Editable` y que sobre éste podíamos hacer un `toString()`. Pero con esta solución estamos perdiendo todo el formato del texto, por lo que no podríamos por ejemplo salvarlo a una base de datos.

La solución a esto último pasa obviamente por recuperar directamente el objeto `Editable` y serializarlo de algún modo, mejor aún si es en un formato estándar. Pues bien, en Android este trabajo ya nos viene hecho de fábrica a través de la clase `Html` [API], que dispone de métodos para convertir cualquier objeto `Spanned` en su representación HTML equivalente. Veamos cómo. Recuperemos el texto de la ventana anterior y utilicemos el método `Html.toHtml(Spannable)` para convertirlo a formato HTML:

```
//Obtiene el texto del control con etiquetas de formato HTML
String aux2 = Html.toHtml(txtTexto.getText());
```

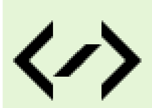
Haciendo esto, obtendríamos una cadena de texto como la siguiente, que ya podríamos por ejemplo almacenar en una base de datos o publicar en cualquier web sin perder el formato de texto establecido:

```
<p>Esto es un <b>simulacro</b>.</p>
```

La operación contraria también es posible, es decir, cargar un cuadro de texto de Android (`EditText`) o una etiqueta (`TextView`) a partir de un fragmento de texto en formato HTML. Para ello podemos utilizar el método `Html.fromHtml(String)` de la siguiente forma:

```
//Asigna texto con formato HTML
txtTexto.setText(
Html.fromHtml("<p>Esto es un <b>simulacro</b>.</p>"),
BufferType.SPANNABLE);
```

Desgraciadamente, aunque es de agradecer que este trabajo venga hecho de casa, hay que decir que tan sólo funciona de forma completa con las opciones de formato más básicas, como negritas, cursivas, subrayado o colores de texto, quedando no soportadas otras sorprendentemente básicas como el tamaño del texto, que aunque sí es correctamente traducido por el método `toHtml()`, es descartado por el método contrario `fromHtml()`. Sí se soporta la inclusión de imágenes, aunque esto lo dejamos para más adelante ya que requiere algo más de explicación.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-imagen-texto

Checkboxes y RadioButtons

Tras hablar de varios de los controles indispensables en cualquier aplicación Android, como son los botones

y los cuadros de texto, en este nuevo apartado vamos a ver cómo utilizar otros dos tipos de controles básicos en muchas aplicaciones, los *checkboxes* y los *radio buttons*.

Control CheckBox [API]

Un control *checkbox* se suele utilizar para marcar o desmarcar opciones en una aplicación, y en Android está representado por la clase del mismo nombre, `CheckBox`. La forma de definirlo en nuestra interfaz y los métodos disponibles para manipularlos desde nuestro código son análogos a los ya comentados para el control `ToggleButton`. De esta forma, para definir un control de este tipo en nuestro layout podemos utilizar el código siguiente, que define un checkbox con el texto "*Márcame*":

```
<CheckBox android:id="@+id/ChkMarcame"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/marcame"
android:checked="false" />
```

En cuanto a la personalización del control podemos decir que éste extiende [indirectamente] del control `TextView`, por lo que todas las opciones de formato ya comentadas en capítulos anteriores son válidas también para este control. Además, podremos utilizar la propiedad `android:checked` para inicializar el estado del control a marcado (`true`) o desmarcado (`false`). Si no establecemos esta propiedad el control aparecerá por defecto en estado desmarcado.

En el código de la aplicación podremos hacer uso de los métodos `isChecked()` para conocer el estado del control, y `setChecked(estado)` para establecer un estado concreto para el control.

```
if (checkBox.isChecked()) {
    checkBox.setChecked(false);
}
```

En cuanto a los posibles eventos que puede lanzar este control, el más interesante es, al margen del siempre válido `onClick`, el que informa de que ha cambiado el estado del control, que recibe el nombre de `onCheckedChanged`. Para implementar las acciones de este evento podríamos utilizar la siguiente lógica, donde tras capturar el evento, y dependiendo del nuevo estado del control (variable `isChecked` recibida como parámetro), haremos una acción u otra:

```
private CheckBox cbMarcame;

//...

cbMarcame = (CheckBox)findViewById(R.id.chkMarcame);

cbMarcame.setOnCheckedChangeListener(
new CheckBox.OnCheckedChangeListener() {

              public void onCheckedChanged(CompoundButton buttonView,
                                          boolean isChecked) {

                    if (isChecked) {
                        cbMarcame.setText("Checkbox marcado!");
                    }
                    else {
                        cbMarcame.setText("Checkbox desmarcado!");
                    }
              }
        });
```

Control RadioButton [API]

Al igual que los controles *checkbox*, un *radio button* puede estar marcado o desmarcado, pero en este caso suelen utilizarse dentro de un grupo de opciones donde una, y sólo una, de ellas debe estar marcada obligatoriamente, es decir, que si se marca una de las opciones se desmarcará automáticamente la que estuviera activa anteriormente. En Android, un grupo de botones `RadioButton` se define mediante un elemento `RadioGroup`, que a su vez contendrá todos los elementos `RadioButton` necesarios. Veamos un ejemplo de cómo definir un grupo de dos controles `RadioButton` en nuestra interfaz:

```
<RadioGroup android:id="@+id/gruporb"
      android:orientation="vertical"
      android:layout_width="match_parent"
      android:layout_height="match_parent" >

      <RadioButton android:id="@+id/radio1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/opcion_1" />

      <RadioButton android:id="@+id/radio2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/opcion_2" />
</RadioGroup>
```

En primer lugar vemos cómo podemos definir el grupo de controles indicando su orientación (vertical u horizontal) al igual que ocurría por ejemplo con un `LinearLayout`. Tras esto, se añaden todos los objetos `RadioButton` necesarios indicando su ID mediante la propiedad `android:id` y su texto mediante `android:text`.

Una vez definida la interfaz podremos manipular el control desde nuestro código java haciendo uso de los diferentes métodos del control `RadioGroup`, los más importantes: `check(id)` para marcar una opción determinada mediante su ID, `clearCheck()` para desmarcar todas las opciones, y `getCheckedRadioButtonId()` que como su nombre indica devolverá el ID de la opción marcada (o el valor -1 si no hay ninguna marcada). Veamos un ejemplo:

```
RadioGroup rg = (RadioGroup)findViewById(R.id.gruporb);
rg.clearCheck();
rg.check(R.id.radio1);
int idSeleccionado = rg.getCheckedRadioButtonId();
```

En cuanto a los eventos lanzados, a parte de `onClick`, al igual que en el caso de los checkboxes, el más importante será el que informa de los cambios en el elemento seleccionado, llamado también en este caso `onCheckedChange`.

Vemos cómo tratar este evento del objeto `RadioGroup`, haciendo por ejemplo que una etiqueta de texto cambie de valor al seleccionar cada opción:

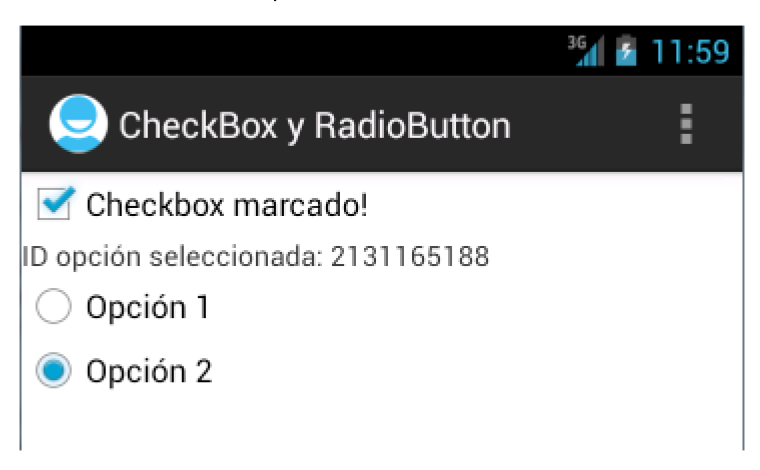
```
private TextView lblMensaje;
private RadioGroup rgOpciones;

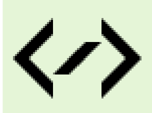
//...

lblMensaje = (TextView)findViewById(R.id.LblSeleccion);
rgOpciones = (RadioGroup)findViewById(R.id.gruporb);
```

```
rgOpciones.setOnCheckedChangeListener(
new RadioGroup.OnCheckedChangeListener() {
            public void onCheckedChanged(RadioGroup group, int checkedId) {
                 lblMensaje.setText("ID opción seleccionada: " + checkedId);
            }
      });
```

Veamos finalmente una imagen del aspecto de estos dos nuevos tipos de controles básicos que hemos comentado en este apartado:





Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-check-radio

Listas Desplegables

Una vez repasados los controles básicos que podemos utilizar en nuestras aplicaciones Android, vamos a dedicar los próximos apartados a describir los diferentes controles de selección disponibles en la plataforma. Al igual que en otros frameworks Android dispone de diversos controles que nos permiten seleccionar una opción dentro de una lista de posibilidades. Así, podremos utilizar listas desplegables (`Spinner`), listas fijas (`ListView`), tablas (`GridView`) y otros controles específicos de la plataforma como por ejemplo las galerías de imágenes (`Gallery`).

En este primer tema dedicado a los controles de selección vamos a describir un elemento importante y común a todos ellos, los adaptadores, y lo vamos a aplicar al primer control de los indicados, las listas desplegables.

Adaptadores en Android (adapters)

Para los desarrolladores de java que hayan utilizado frameworks de interfaz gráfica como Swing, el concepto de adaptador les resultará familiar. Un adaptador representa algo así como una interfaz común al modelo de datos que existe por detrás de todos los controles de selección que hemos comentado. Dicho de otra forma, todos los controles de selección accederán a los datos que contienen a través de un adaptador.

Además de proveer de datos a los controles visuales, el adaptador también será responsable de generar a partir de estos datos las vistas específicas que se mostrarán dentro del control de selección. Por ejemplo, si cada elemento de una lista estuviera formado a su vez por una imagen y varias etiquetas, el responsable de generar y establecer el contenido de todos estos "sub-elementos" a partir de los datos será el propio adaptador.

Android proporciona de serie varios tipos de adaptadores sencillos, aunque podemos extender su funcionalidad fácilmente para adaptarlos a nuestras necesidades. Los más comunes son los siguientes:

- ArrayAdapter. Es el más sencillo de todos los adaptadores, y provee de datos a un control de selección a partir de un array de objetos de cualquier tipo.

- SimpleAdapter. Se utiliza para mapear datos sobre los diferentes controles definidos en un fichero XML de layout.

- SimpleCursorAdapter. Se utiliza para mapear las columnas de un cursor abierto sobre una base de datos sobre los diferentes elementos visuales contenidos en el control de selección.

Para no complicar excesivamente los tutoriales, por ahora nos vamos a conformar con describir la forma de utilizar un ArrayAdapter con los diferentes controles de selección disponibles. Más adelante aprenderemos a utilizar el resto de adaptadores en contextos más específicos.

Veamos cómo crear un adaptador de tipo ArrayAdapter para trabajar con un array genérico de java:

```
final String[] datos =
        new String[]{"Elem1","Elem2","Elem3","Elem4","Elem5"};

ArrayAdapter<String> adaptador =
        new ArrayAdapter<String>(this,
            android.R.layout.simple_spinner_item, datos);
```

Comentemos un poco el código. Sobre la primera línea no hay nada que decir, es tan sólo la definición del array java que contendrá los datos a mostrar en el control, en este caso un array sencillo con cinco cadenas de caracteres. En la segunda línea creamos el adaptador en sí, al que pasamos 3 parámetros:

1. El contexto, que normalmente será simplemente una referencia a la actividad donde se crea el adaptador.

2. El ID del layout sobre el que se mostrarán los datos del control. En este caso le pasamos el ID de un layout predefinido en Android (android.R.layout.simple_spinner_item), formado únicamente por un control TextView, pero podríamos pasarle el ID de cualquier layout personalizado de nuestro proyecto con cualquier estructura y conjunto de controles, más adelante veremos cómo (en el apartado dedicado a las listas fijas).

3. El array que contiene los datos a mostrar.

Con esto ya tendríamos creado nuestro adaptador para los datos a mostrar y ya tan sólo nos quedaría asignar este adaptador a nuestro control de selección para que éste mostrase los datos en la aplicación.

Una alternativa a tener en cuenta si los datos a mostrar en el control son estáticos sería definir la lista de posibles valores como un recurso de tipo string-array . Para ello, primero crearíamos un nuevo fichero XML en la carpeta /res/values llamado por ejemplo valores_array.xml e incluiríamos en él los valores seleccionables de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="valores_array">
        <item>Elem1</item>
        <item>Elem2</item>
        <item>Elem3</item>
        <item>Elem4</item>
        <item>Elem5</item>
    </string-array>
</resources>
```

Tras esto, a la hora de crear el adaptador, utilizaríamos el método createFromResource() para hacer referencia a este array XML que acabamos de crear:

```
ArrayAdapter<CharSequence> adapter =
    ArrayAdapter.createFromResource(this,
        R.array.valores_array, android.R.layout.simple_spinner_item);
```

Control Spinner [API]

Las listas desplegables en Android se llaman `Spinner`. Funcionan de forma similar al de cualquier control de este tipo, el usuario selecciona la lista, se muestra una especie de lista emergente al usuario con todas las opciones disponibles y al seleccionarse una de ellas ésta queda fijada en el control. Para añadir una lista de este tipo a nuestra aplicación podemos utilizar el código siguiente:

```
<Spinner android:id="@+id/CmbOpciones"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

Poco vamos a comentar de aquí ya que lo que nos interesan realmente son los datos a mostrar. En cualquier caso, las opciones para personalizar el aspecto visual del control (fondo, color y tamaño de fuente, etc) son las mismas ya comentadas para los controles básicos.

Para enlazar nuestro adaptador (y por tanto nuestros datos) a este control utilizaremos el siguiente código java:

```
private Spinner cmbOpciones;

//...

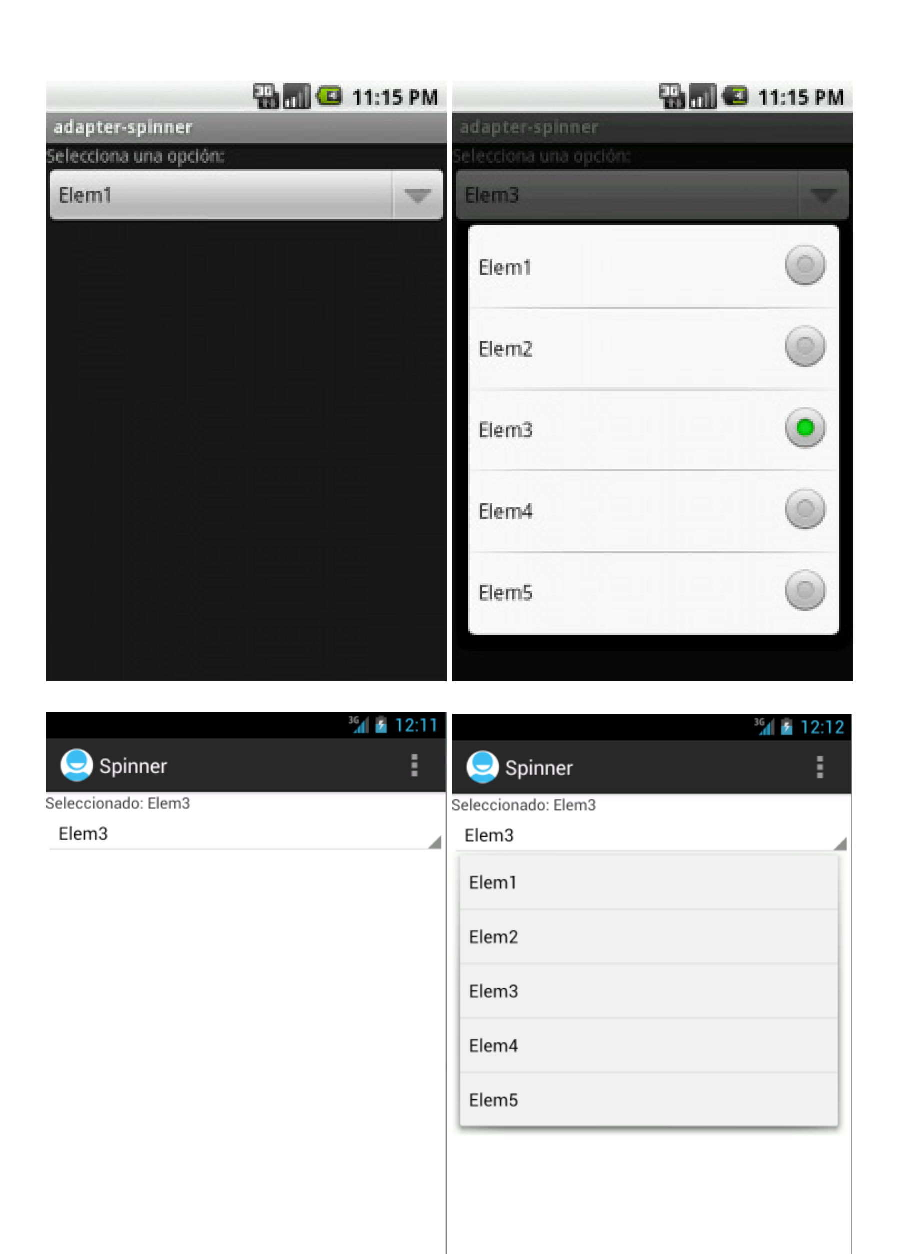
cmbOpciones = (Spinner)findViewById(R.id.CmbOpciones);

adaptador.setDropDownViewResource(
        android.R.layout.simple_spinner_dropdown_item);

cmbOpciones.setAdapter(adaptador);
```

Comenzamos como siempre por obtener una referencia al control a través de su ID. Y en la última línea asignamos el adaptador al control mediante el método `setAdapter()`. ¿Y la segunda línea para qué es? Cuando indicamos en el apartado anterior cómo construir un adaptador vimos cómo uno de los parámetros que le pasábamos era el ID del layout que utilizaríamos para visualizar los elementos del control. Sin embargo, en el caso del control `Spinner`, este layout tan sólo se aplicará al elemento seleccionado en la lista, es decir, al que se muestra directamente sobre el propio control cuando no está desplegado. Sin embargo, antes indicamos que el funcionamiento normal del control `Spinner` incluye entre otras cosas mostrar una lista emergente con todas las opciones disponibles. Pues bien, para personalizar también el aspecto de cada elemento en dicha lista emergente tenemos el método `setDropDownViewResource(ID_layout)`, al que podemos pasar otro ID de layout distinto al primero sobre el que se mostrarán los elementos de la lista emergente. En este caso hemos utilizado otro layout predefinido en Android para las listas desplegables (`android.R.layout.simple_spinner_dropdown_item`), formado por una etiqueta de texto con la descripción de la opción y un marcador circular a la derecha que indica si la opción está o no seleccionada (esto último sólo para Android 2.x).

Con estas simples líneas de código conseguiremos mostrar un control como el que vemos en las siguientes imágenes, tanto para Android 2.x como para la versión 4.x



Como se puede observar en las imágenes, la representación del elemento seleccionado (primera imagen) y el de las opciones disponibles (segunda imagen) es distinto, incluyendo el segundo de ellos incluso algún elemento gráfico a la derecha para mostrar el estado de cada opción (en este caso, sólo para Android 2). Como hemos comentado, esto es debido a la utilización de dos layouts diferentes para uno y otros elementos.

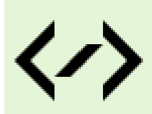
En cuanto a los eventos lanzados por el control `Spinner`, el más comúnmente utilizado será el generado al seleccionarse una opción de la lista desplegable, `onItemSelected`.

Para capturar este evento se procederá de forma similar a lo ya visto para otros controles anteriormente, asignándole su controlador mediante el método `setOnItemSelectedListener()`:

```
cmbOpciones.setOnItemSelectedListener(
        new AdapterView.OnItemSelectedListener() {
            public void onItemSelected(AdapterView<?> parent,
                android.view.View v, int position, long id) {
                    lblMensaje.setText("Seleccionado: " + datos[position]);
            }


            public void onNothingSelected(AdapterView<?> parent) {
                lblMensaje.setText("");
            }
});
```

A diferencia de ocasiones anteriores, para este evento definimos dos métodos, el primero de ellos (`onItemSelected`) será llamado cada vez que se selecciones una opción en la lista desplegable, y el segundo (`onNothingSelected`) que se llamará cuando no haya ninguna opción seleccionada (esto puede ocurrir por ejemplo si el adaptador no tiene datos).



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-spinner

Listas

En el apartado anterior ya comenzamos a hablar de los controles de selección en Android, empezando por explicar el concepto de adaptador y describiendo el control `Spinner`. En este nuevo apartado nos vamos a centrar en el control de selección más utilizado de todos, el `ListView`.

Un control `ListView` muestra al usuario una lista de opciones seleccionables directamente sobre el propio control, sin listas emergentes como en el caso del control `Spinner`. En caso de existir más opciones de las que se pueden mostrar sobre el control se podrá por supuesto hacer *scroll* sobre la lista para acceder al resto de elementos.

Veamos primero cómo podemos añadir un control `ListView` a nuestra interfaz de usuario:

```
<ListView android:id="@+id/LstOpciones"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

Una vez más, podremos modificar el aspecto del control utilizando las propiedades de fuente y color ya comentadas en capítulos anteriores. Por su parte, para enlazar los datos con el control podemos utilizar por ejemplo el mismo código que ya vimos para el control `Spinner`. Definiremos primero un array con nuestros datos de prueba, crearemos posteriormente el adaptador de tipo `ArrayAdapter` y lo asignaremos finalmente al control mediante el método `setAdapter()`:

```
final String[] datos =
    new String[]{"Elem1","Elem2","Elem3","Elem4","Elem5"};

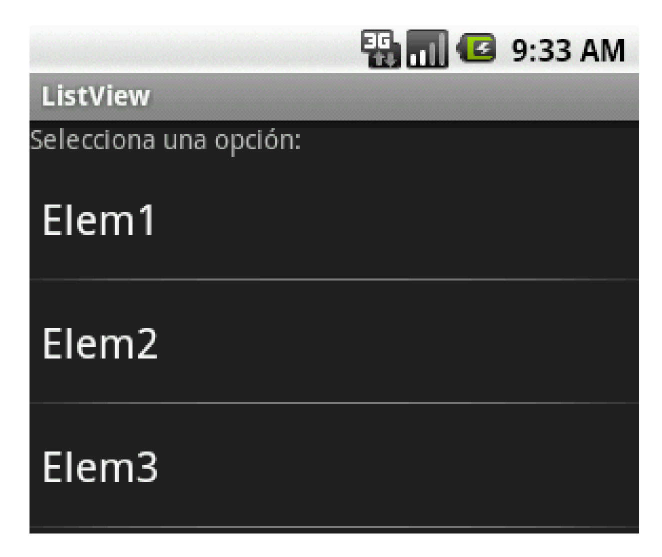
ArrayAdapter<String> adaptador =
    new ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1, datos);

lstOpciones = (ListView)findViewById(R.id.LstOpciones);

lstOpciones.setAdapter(adaptador);
```

NOTA: En caso de necesitar mostrar en la lista datos procedentes de una base de datos la mejor práctica es utilizar un `Loader` (concretamente un `CursorLoader`), que cargará los datos de forma asíncrona de forma que la aplicación no se bloquee durante la carga. Esto lo veremos más adelante en el curso, ahora nos conformaremos con cargar datos estáticos procedentes de un array.

En el código anterior, para mostrar los datos de cada elemento hemos utilizado otro layout genérico de Android para los controles de tipo `ListView` (`android.R.layout.simple_list_item_1`), formado únicamente por un `TextView` con unas dimensiones determinadas. La lista creada quedaría como se muestra en la imagen siguiente (por ejemplo para Android 2.x, aunque sería prácticamente igual en versiones más recientes):



Como podéis comprobar el uso básico del control `ListView` es completamente análogo al ya comentado para el control `Spinner`.

Hasta aquí todo sencillo. Pero, ¿y si necesitamos mostrar datos más complejos en la lista? ¿qué ocurre si necesitamos que cada elemento de la lista esté formado a su vez por varios elementos? Pues vamos a provechar este capítulo dedicado a los `ListView` para ver cómo podríamos conseguirlo, aunque todo lo que comentaré es extensible a otros controles de selección.

Para no complicar mucho el tema vamos a hacer que cada elemento de la lista muestre por ejemplo dos líneas de texto a modo de título y subtítulo con formatos diferentes (por supuesto se podrían añadir muchos más elementos, por ejemplo imágenes, checkboxes, etc).

En primer lugar vamos a crear una nueva clase java para contener nuestros datos de prueba. Vamos a llamarla `Titular` y tan sólo va a contener dos atributos, título y subtítulo.

```
package net.sgoliver;

public class Titular
{
    private String titulo;
    private String subtitulo;

    public Titular(String tit, String sub){
        titulo = tit;
        subtitulo = sub;
    }

    public String getTitulo(){
        return titulo;
    }

    public String getSubtitulo(){
        return subtitulo;
    }
}
```

En cada elemento de la lista queremos mostrar ambos datos, por lo que el siguiente paso será crear un layout XML con la estructura que deseemos. En mi caso voy a mostrarlos en dos etiquetas de texto (`TextView`), la primera de ellas en negrita y con un tamaño de letra un poco mayor.

Llamaremos a este layout "`listitem_titular.xml`":

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:orientation="vertical">

<TextView android:id="@+id/LblTitulo"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textStyle="bold"
    android:textSize="20px" />

<TextView android:id="@+id/LblSubTitulo"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textStyle="normal"
    android:textSize="12px" />

</LinearLayout>
```

Ahora que ya tenemos creados tanto el soporte para nuestros datos como el layout que necesitamos para visualizarlos, lo siguiente que debemos hacer será indicarle al adaptador cómo debe utilizar ambas cosas para generar nuestra interfaz de usuario final. Para ello vamos a crear nuestro propio adaptador extendiendo de la clase `ArrayAdapter`.

```
class AdaptadorTitulares extends ArrayAdapter {

    Activity context;

        AdaptadorTitulares(Activity context) {
            super(context, R.layout.listitem_titular, datos);
            this.context = context;
        }

        public View getView(int position,View convertView,ViewGroup parent) {
            LayoutInflater inflater = context.getLayoutInflater();
            View item = inflater.inflate(R.layout.listitem_titular, null);

            TextView lblTitulo = (TextView)item.findViewById(R.id.LblTitulo);
            lblTitulo.setText(datos[position].getTitulo());

            TextView lblSubtitulo =
                (TextView)item.findViewById(R.id.LblSubTitulo);

            lblSubtitulo.setText(datos[position].getSubtitulo());

            return(item);
      }
}
```

Analicemos el código anterior. Lo primero que encontramos es el constructor para nuestro adaptador, al que sólo pasaremos el contexto (que será la actividad desde la que se crea el adaptador). En este constructor tan sólo guardaremos el contexto para nuestro uso posterior y llamaremos al constructor padre tal como ya vimos al principio de este capítulo, pasándole el ID del layout que queremos utilizar (en nuestro caso el nuevo que hemos creado, `listitem_titular`) y el array que contiene los datos a mostrar.

Posteriormente, redefinimos el método encargado de generar y rellenar con nuestros datos todos los controles necesarios de la interfaz gráfica de cada elemento de la lista. Este método es `getView()`.

El método `getView()` se llamará cada vez que haya que mostrar un elemento de la lista. Lo primero que debe hacer es "inflar" el layout XML que hemos creado. Esto consiste en consultar el XML de nuestro layout y crear e inicializar la estructura de objetos java equivalente. Para ello, crearemos un nuevo objeto `LayoutInflater` y generaremos la estructura de objetos mediante su método `inflate(id_layout)`.

Tras esto, tan sólo tendremos que obtener la referencia a cada una de nuestras etiquetas como siempre lo hemos hecho y asignar su texto correspondiente según los datos de nuestro array y la posición del elemento actual (parámetro position del método `getView()`).

Una vez tenemos definido el comportamiento de nuestro adaptador la forma de proceder en la actividad principal será análoga a lo ya comentado, definiremos el array de datos de prueba, crearemos el adaptador y lo asignaremos al control mediante `setAdapter()`:

```
private Titular[] datos =
    new Titular[]{
        new Titular("Título 1", "Subtítulo largo 1"),
        new Titular("Título 2", "Subtítulo largo 2"),
        new Titular("Título 3", "Subtítulo largo 3"),
        new Titular("Título 4", "Subtítulo largo 4"),
        new Titular("Título 5", "Subtítulo largo 5")};

//...
```

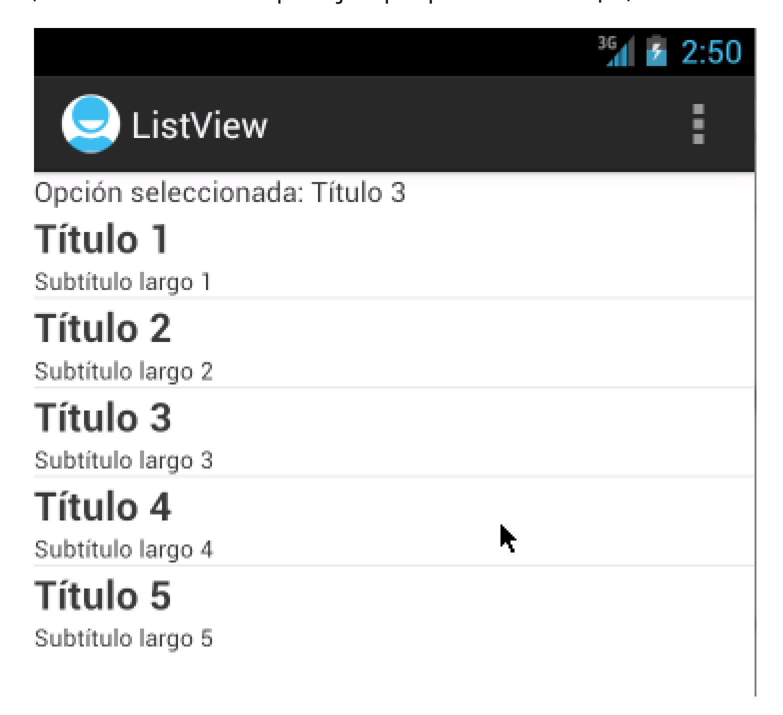
```
//...

AdaptadorTitulares adaptador =
        new AdaptadorTitulares(this);

lstOpciones = (ListView)findViewById(R.id.LstOpciones);

lstOpciones.setAdapter(adaptador);
```

Hecho esto, y si todo ha ido bien, nuestra nueva lista debería quedar como vemos en la imagen siguiente (esta vez lo muestro por ejemplo para Android 4.x):



Por último comentemos un poco los eventos de este tipo de controles. Si quisiéramos realizar cualquier acción al pulsarse sobre un elemento de la lista creada tendremos que implementar el evento `onItemClick`. Veamos cómo con un ejemplo:

```
lstOpciones.setOnItemClickListener(new OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> a, View v, int position, long id) {
        //Alternativa 1:
        String opcionSeleccionada =
            ((Titular)a.getAdapter().getItem(position)).getTitulo();

       //Alternativa 2:
       //String opcionSeleccionada =
       //          ((TextView)v.findViewById(R.id.LblTitulo))
       //                .getText().toString();

       lblEtiqueta.setText("Opción seleccionada: " + opcionSeleccionada);
    }
});
```

Este evento recibe 4 parámetros:

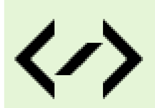
- Referencia al control lista que ha recibido el click (`AdapterView a`).

- Referencia al objeto View correspondiente al ítem pulsado de la lista (`View v`).
- Posición del elemento pulsado dentro del adaptador de la lista (`int position`).
- Id del elemento pulsado (`long id`).

Con todos estos datos, si quisiéramos por ejemplo mostrar el título de la opción pulsada en la etiqueta de texto superior (`lblEtiqueta`) tendríamos dos posibilidades:

1. Acceder al adaptador de la lista mediante el método `getAdapter()` y a partir de éste obtener mediante `getItem()` el elemento cuya posición sea `position`. Esto nos devolvería un objeto de tipo `Titular`, por lo que obtendríamos el título llamando a su método `getTitulo()`.
2. Acceder directamente a la vista que se ha pulsado, que tendría la estructura definida en nuestro layout personalizado `listitem_titular.xml`, y obtener mediante `findViewById()` y `getText()` el texto del control que alberga el campo título.

Y esto sería todo por ahora. Aunque ya sabemos utilizar y personalizar las listas en Android, en el próximo apartado daremos algunas indicaciones para utilizar de una forma mucho más eficiente los controles de este tipo, algo que los usuarios de nuestra aplicación agradecerán enormemente al mejorarse la respuesta de la aplicación y reducirse el consumo de batería.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-listview

Optimización de listas

En el apartado anterior ya vimos cómo utilizar los controles de tipo `ListView` en Android. Sin embargo, acabamos comentando que existía una forma más eficiente de hacer uso de dicho control, de forma que la respuesta de nuestra aplicación fuera más ágil y se redujese el consumo de batería, algo que en plataformas móviles siempre es importante.

Como base para este tema vamos a utilizar como código el que ya escribimos en el capítulo anterior, por lo que si has llegado hasta aquí directamente te recomiendo que leas primero el primer post dedicado al control `ListView`.

Cuando comentamos cómo crear nuestro propio adaptador, extendiendo de `ArrayAdapter`, para personalizar la forma en que nuestros datos se iban a mostrar en la lista escribimos el siguiente código:

```
class AdaptadorTitulares extends ArrayAdapter {
    Activity context;

    AdaptadorTitulares(Activity context) {
        super(context, R.layout.listitem_titular, datos);
        this.context = context;
    }

    public View getView(int position, View convertView, ViewGroup parent) {
        LayoutInflater inflater = context.getLayoutInflater();
        View item = inflater.inflate(R.layout.listitem_titular, null);
```

```
        TextView lblTitulo = (TextView)item.findViewById(R.id.LblTitulo);
        lblTitulo.setText(datos[position].getTitulo());

        TextView lblSubtitulo =
                    (TextView)item.findViewById(R.id.LblSubTitulo);
        lblSubtitulo.setText(datos[position].getSubtitulo());

        return(item);
    }
}
```

Centrándonos en la definición del método `getView()` vimos que la forma normal de proceder consistía en primer lugar en "inflar" nuestro layout XML personalizado para crear todos los objetos correspondientes (con la estructura descrita en el XML) y posteriormente acceder a dichos objetos para modificar sus propiedades. Sin embargo, hay que tener en cuenta que esto se hace todas y cada una de las veces que se necesita mostrar un elemento de la lista en pantalla, se haya mostrado ya o no con anterioridad, ya que Android no "guarda" los elementos de la lista que desaparecen de pantalla (por ejemplo al hacer *scroll* sobre la lista). El efecto de esto es obvio, dependiendo del tamaño de la lista y sobre todo de la complejidad del layout que hayamos definido esto puede suponer la creación y destrucción de cantidades ingentes de objetos (que puede que ni siquiera nos sean necesarios), es decir, que la acción de inflar un layout XML puede ser bastante costosa, lo que podría aumentar mucho, y sin necesidad, el uso de CPU, de memoria, y de batería.

Para aliviar este problema, Android nos propone un método que permite reutilizar algún layout que ya hayamos inflado con anterioridad y que ya no nos haga falta por algún motivo, por ejemplo porque el elemento correspondiente de la lista ha desaparecido de la pantalla al hacer *scroll*. De esta forma evitamos todo el trabajo de crear y estructurar todos los objetos asociados al layout, por lo que tan sólo nos quedaría obtener la referencia a ellos mediante `findViewById()` y modificar sus propiedades.

¿Pero cómo podemos reutilizar estos layouts "obsoletos"? Pues es bien sencillo, siempre que exista algún layout que pueda ser reutilizado éste se va a recibir a través del parámetro `convertView` del método `getView()`. De esta forma, en los casos en que éste no sea `null` podremos obviar el trabajo de inflar el layout. Veamos cómo quedaría el método `getView()` tras esta optimización:

```
public View getView(int position, View convertView, ViewGroup parent)
{
    View item = convertView;

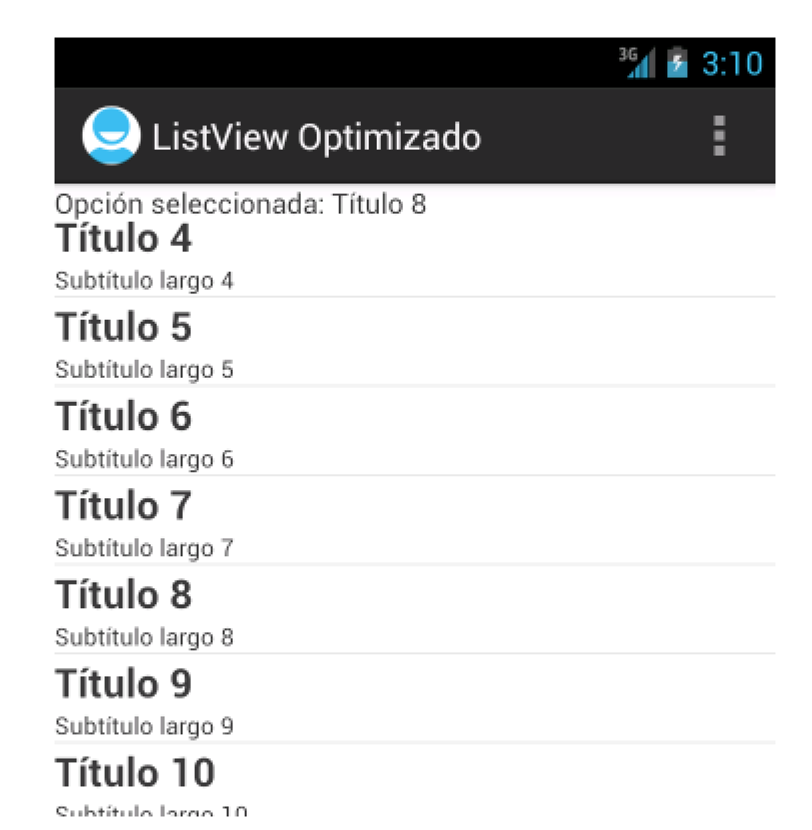
    if(item == null)
    {
        LayoutInflater inflater = context.getLayoutInflater();
        item = inflater.inflate(R.layout.listitem_titular, null);
    }

    TextView lblTitulo = (TextView)item.findViewById(R.id.LblTitulo);
    lblTitulo.setText(datos[position].getTitulo());

    TextView lblSubtitulo = (TextView)item.findViewById(R.id.LblSubTitulo);
    lblSubtitulo.setText(datos[position].getSubtitulo());

    return(item);
}
```

Si añadimos más elementos a la lista y ejecutamos ahora la aplicación podemos comprobar que al hacer scroll sobre la lista todo sigue funcionando con normalidad, con la diferencia de que le estamos ahorrando gran cantidad de trabajo a la CPU.



Pero vamos a ir un poco más allá. Con la optimización que acabamos de implementar conseguimos ahorrarnos el trabajo de inflar el layout definido cada vez que se muestra un nuevo elemento. Pero aún hay otras dos llamadas relativamente costosas que se siguen ejecutando en todas las llamadas. Me refiero a la obtención de la referencia a cada uno de los objetos a modificar mediante el método `findViewById()`. La búsqueda por ID de un control determinado dentro del árbol de objetos de un layout también puede ser una tarea costosa dependiendo de la complejidad del propio layout. ¿Por qué no aprovechamos que estamos "guardando" un layout anterior para guardar también la referencia a los controles que lo forman de forma que no tengamos que volver a buscarlos? Pues eso es exactamente lo que vamos a hacer mediante lo que en los ejemplos de Android llaman un `ViewHolder`. La clase `ViewHolder` tan sólo va a contener una referencia a cada uno de los controles que tengamos que manipular de nuestro layout, en nuestro caso las dos etiquetas de texto. Definamos por tanto esta clase de la siguiente forma:

```
static class ViewHolder {
    TextView titulo;
    TextView subtitulo; }
```

La idea será por tanto crear e inicializar el objeto `ViewHolder` la primera vez que inflemos un elemento de la lista y asociarlo a dicho elemento de forma que posteriormente podamos recuperarlo fácilmente. ¿Pero dónde lo guardamos? Fácil, en Android todos los controles tienen una propiedad llamada `Tag` (podemos asignarla y recuperarla mediante los métodos `setTag()` y `getTag()` respectivamente) que puede contener cualquier tipo de objeto, por lo que resulta ideal para guardar nuestro objeto `ViewHolder`. De esta forma, cuando el parámetro `convertView` llegue informado sabremos que también tendremos disponibles las referencias a sus controles hijos a través de la propiedad `Tag`. Veamos el código modificado de `getView()` para aprovechar esta nueva optimización:

```
public View getView(int position, View convertView, ViewGroup parent)
{
    View item = convertView;
    ViewHolder holder;
```

```
    if(item == null)
    {
        LayoutInflater inflater = context.getLayoutInflater();
        item = inflater.inflate(R.layout.listitem_titular, null);

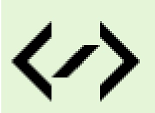
        holder = new ViewHolder();
        holder.titulo = (TextView)item.findViewById(R.id.LblTitulo);
        holder.subtitulo = (TextView)item.findViewById(R.id.LblSubTitulo);

        item.setTag(holder);
    }
    else
    {
        holder = (ViewHolder)item.getTag();
    }

    holder.titulo.setText(datos[position].getTitulo());
    holder.subtitulo.setText(datos[position].getSubtitulo());

    return(item);
}
```

Con estas dos optimizaciones hemos conseguido que la aplicación sea mucho más respetuosa con los recursos del dispositivo de nuestros usuarios, algo que sin duda nos agradecerán.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-listview-opt

Grids

Tras haber visto en apartados anteriores los dos controles de selección más comunes en cualquier interfaz gráfica, como son las listas desplegables (`Spinner`) y las listas "fijas" (`ListView`), tanto en su versión básica como optimizada, en este nuevo apartado vamos a terminar de comentar los controles de selección con otro menos común pero no por ello menos útil, el control `GridView`.

El control `GridView` de Android presenta al usuario un conjunto de opciones seleccionables distribuidas de forma tabular, o dicho de otra forma, divididas en filas y columnas. Dada la naturaleza del control ya podéis imaginar sus propiedades más importantes, que paso a enumerar a continuación:

Propiedad	Descripción
`android:numColumns`	Indica el número de columnas de la tabla o "`auto_fit`" si queremos que sea calculado por el propio sistema operativo a partir de las siguientes propiedades.
`android:columnWidth`	Indica el ancho de las columnas de la tabla.
`android:horizontalSpacing`	Indica el espacio horizontal entre celdas.
`android:verticalSpacing`	Indica el espacio vertical entre celdas.
`android:stretchMode`	Indica qué hacer con el espacio horizontal sobrante. Si se establece al valor `columnWidth` este espacio será absorbido a partes iguales por las columnas de la tabla. Si por el contrario se establece a `spacingWidth` será absorbido a partes iguales por los espacios entre celdas.

Veamos cómo definiríamos un GridView de ejemplo en nuestra aplicación:

```
<GridView android:id="@+id/GridOpciones"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:numColumns="auto_fit"
    android:columnWidth="80px"
    android:horizontalSpacing="5dp"
    android:verticalSpacing="10dp"
    android:stretchMode="columnWidth" />
```

Una vez definida la interfaz de usuario, la forma de asignar los datos desde el código de la aplicación es completamente análoga a la ya comentada tanto para las listas desplegables como para las listas estáticas: creamos un array genérico que contenga nuestros datos de prueba, declaramos un adaptador de tipo ArrayAdapter (como ya comentamos, si los datos proceden de una base de datos lo normal será utilizar un SimpleCursorAdapter, pero de eso nos ocuparemos más adelante en el curso) pasándole en este caso un layout genérico (simple_list_item_1, compuesto por un simple TextView) y asociamos el adaptador al control GridView mediante su método setAdapter():

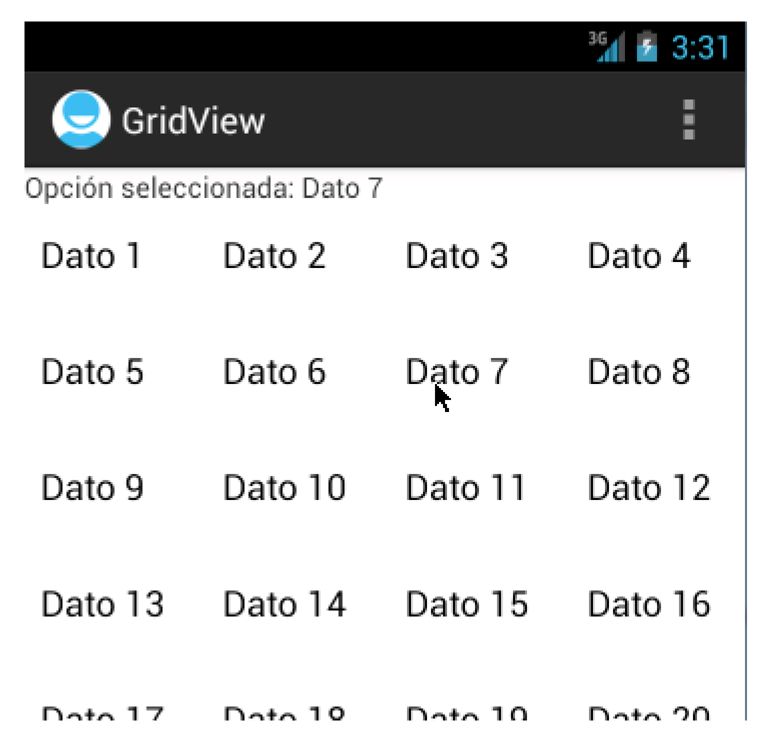
```
private String[] datos = new String[25];
//...
for(int i=1; i<=25; i++)
        datos[i-1] = "Dato " + i;

ArrayAdapter<String> adaptador =
   new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, datos);

grdOpciones = (GridView)findViewById(R.id.GridOpciones);

grdOpciones.setAdapter(adaptador);
```

Por defecto, los datos del *array* se añadirán al control GridView ordenados por filas, y por supuesto, si no caben todos en la pantalla se podrá hacer *scroll* sobre la tabla. Vemos en una imagen cómo queda nuestra aplicación de prueba:

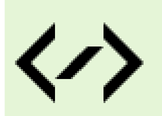


En cuanto a los eventos disponibles, el más interesante vuelve a ser el lanzado al seleccionarse una celda determinada de la tabla: onItemClick. Este evento podemos capturarlo de la misma forma que hacíamos con los controles Spinner y ListView. Veamos un ejemplo de cómo hacerlo:

```
grdOpciones.setOnItemClickListener(
   new AdapterView.OnItemClickListener() {
      public void onItemClick(AdapterView<?> parent,
         android.view.View v, int position, long id) {
            lblMensaje.setText("Opción seleccionada: " + datos[position]);
         }
   });
```

Todo lo comentado hasta el momento se refiere al uso básico del control GridView, pero por supuesto podríamos aplicar de forma prácticamente directa todo lo comentado para las listas en los dos apartados anteriores, es decir, la personalización de las celdas para presentar datos complejos creando nuestro propio adaptador, y las distintas optimizaciones para mejorar el rendimiento de la aplicación y el gasto de batería.

Y con esto finalizamos todo lo que tenía previsto contar sobre los distintos controles disponibles "de serie" en Android para construir nuestras interfaces de usuario. Existen muchos más, pero por el momento nos vamos a conformar con los ya vistos, que serán los utilizados con más frecuencia.



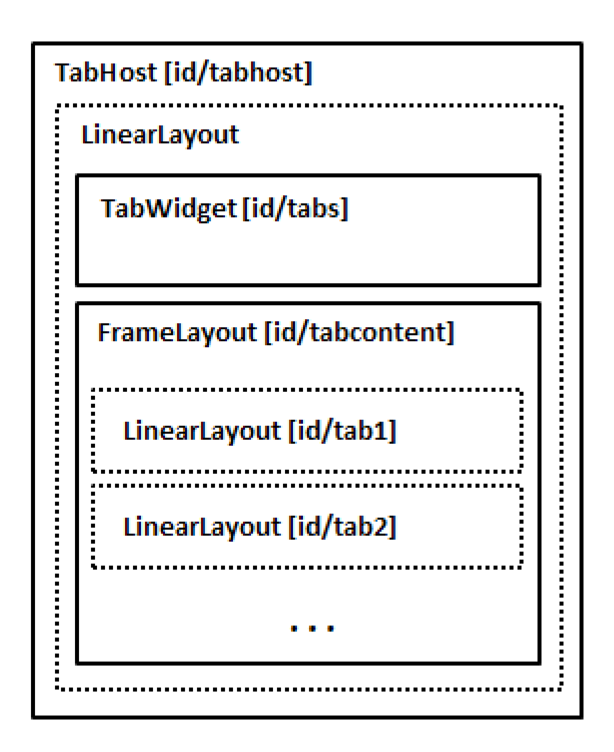
Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-gridview

Pestañas

En apartados anteriores hemos visto como dar forma a la interfaz de usuario de nuestra aplicación mediante el uso de diversos tipos de *layouts*, como por ejemplo los lineales, absolutos, relativos, u otros más elaborados como los de tipo lista o tabla. Éstos van a ser siempre los elementos organizativos básicos de nuestra interfaz, pero sin embargo, dado el poco espacio con el que contamos en las pantallas de muchos dispositivos, o simplemente por cuestiones de organización, a veces es necesario/interesante dividir nuestros controles en varias pantallas. Y una de las formas clásicas de conseguir esto consiste en la distribución de los elementos por pestañas o *tabs*. Android por supuesto permite utilizar este tipo de interfaces, aunque lo hace de una forma un tanto peculiar, ya que la implementación no va a depender de un sólo elemento sino de varios, que además deben estar distribuidos y estructurados de una forma determinada nada arbitraria. Adicionalmente no nos bastará simplemente con definir la interfaz en XML como hemos hecho en otras ocasiones, sino que también necesitaremos completar el conjunto con algunas líneas de código. Desarrollemos esto poco a poco.

En Android, el elemento principal de un conjunto de pestañas será el control TabHost. Éste va a ser el contenedor principal de nuestro conjunto de pestañas y deberá tener obligatoriamente como id el valor "@android:id/tabhost". Dentro de éste vamos a incluir un LinearLayout que nos servirá para distribuir verticalmente las secciones principales del layout: la sección de pestañas en la parte superior y la sección de contenido en la parte inferior. La sección de pestañas se representará mediante un elemento TabWidget, que deberá tener como id el valor "@android:id/tabs", y como contenedor para el contenido de las pestañas añadiremos un FrameLayout con el id obligatorio "@android:id/tabcontent". Por último, dentro del FrameLayout incluiremos el contenido de cada pestaña, normalmente cada uno dentro de su propio layout principal (en mi caso he utilizado LinearLayout) y con un id único que nos permita posteriormente hacer referencia a ellos fácilmente (en mi caso he utilizado por ejemplo los ids "*tab1*", "*tab2*", ...). A continuación represento de forma gráfica toda la estructura descrita.



Si traducimos esta estructura a nuestro fichero de layout XML tendríamos lo siguiente:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_height="match_parent"
    android:layout_width="match_parent">

<TabHost android:id="@android:id/tabhost"
             android:layout_width="match_parent"
             android:layout_height="match_parent">

       <LinearLayout
            android:orientation="vertical"
            android:layout_width="match_parent"
            android:layout_height="match_parent" >

           <TabWidget android:layout_width="match_parent"
                  android:layout_height="wrap_content"
                  android:id="@android:id/tabs" />

           <FrameLayout android:layout_width="match_parent"
                          android:layout_height="match_parent"
                          android:id="@android:id/tabcontent" >

            <LinearLayout android:id="@+id/tab1"
                        android:orientation="vertical"
                        android:layout_width="match_parent"
                        android:layout_height="match_parent" >

                  <TextView android:id="@+id/textView1"
                          android:text="Contenido Tab 1"
                          android:layout_width="wrap_content"
                          android:layout_height="wrap_content" />
            </LinearLayout>
```

```
            <LinearLayout android:id="@+id/tab2"
                          android:orientation="vertical"
                          android:layout_width="match_parent"
                          android:layout_height="match_parent" >

                  <TextView android:id="@+id/textView2"
                          android:text="Contenido Tab 2"
                          android:layout_width="wrap_content"
                          android:layout_height="wrap_content" />
            </LinearLayout>

           </FrameLayout>

      </LinearLayout>

</TabHost>

</LinearLayout>
```

Como podéis ver, como contenido de las pestañas tan sólo he añadido por simplicidad una etiqueta de texto con el texto "*Contenido Tab NºTab*". Esto nos permitirá ver que el conjunto de pestañas funciona correctamente cuando ejecutemos la aplicación.

Con esto ya tendríamos montada toda la estructura de controles necesaria para nuestra interfaz de pestañas. Sin embargo, como ya dijimos al principio del apartado, con esto no es suficiente. Necesitamos asociar de alguna forma cada pestaña con su contenido, de forma que el control se comporte correctamente cuando cambiamos de pestaña. Y esto tendremos que hacerlo mediante código en nuestra actividad principal.

Empezaremos obteniendo una referencia al control principal `TabHost` y preparándolo para su configuración llamando a su método `setup()`. Tras esto, crearemos un objeto de tipo `TabSpec` para cada una de las pestañas que queramos añadir mediante el método `newTabSpec()`, al que pasaremos como parámetro una etiqueta identificativa de la pestaña (en mi caso de ejemplo "*mitab1*", "*mitab2*", …). Además, también le asignaremos el layout de contenido correspondiente a la pestaña llamando al método `setContent()`, e indicaremos el texto y el icono que queremos mostrar en la pestaña mediante el método `setIndicator(texto, icono)`. Veamos el código completo.

```
Resources res = getResources();

TabHost tabs=(TabHost)findViewById(android.R.id.tabhost);
tabs.setup();

TabHost.TabSpec spec=tabs.newTabSpec("mitab1");
spec.setContent(R.id.tab1);
spec.setIndicator("TAB1",
                  res.getDrawable(android.R.drawable.ic_btn_speak_now));
tabs.addTab(spec);

spec=tabs.newTabSpec("mitab2");
spec.setContent(R.id.tab2);
spec.setIndicator("TAB2",
                  res.getDrawable(android.R.drawable.ic_dialog_map));
tabs.addTab(spec);

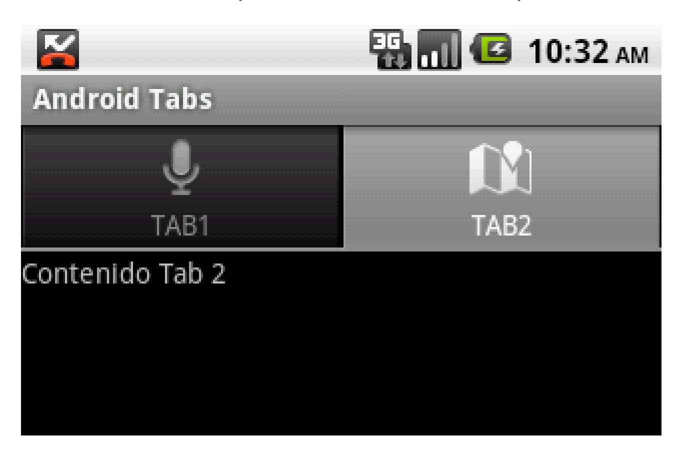
tabs.setCurrentTab(0);
```

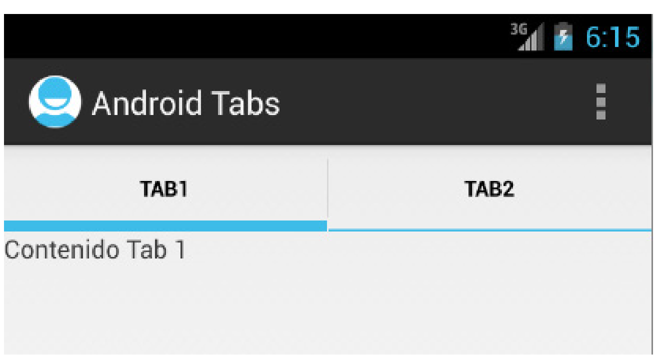
Si vemos el código, vemos por ejemplo como para la primera pestaña creamos un objeto `TabSpec` con la etiqueta "*mitab1*", le asignamos como contenido uno de los `LinearLayout` que incluimos en la sección de contenido (en este caso `R.id.tab1`) y finalmente le asignamos el texto "*TAB1*" y el icono

`android.R.drawable.ic_btn_speak_now` (Éste es un icono incluido con la propia plataforma Android. Si no existiera en vuestra versión podéis sustituirlo por cualquier otro icono). Finalmente añadimos la nueva pestaña al control mediante el método `addTab()`.

Si ejecutamos ahora la aplicación tendremos algo como lo que muestra la siguiente imagen, donde podremos cambiar de pestaña y comprobar cómo se muestra correctamente el contenido de la misma.

Lo muestro tanto para Android 2.x como para Android 4.x

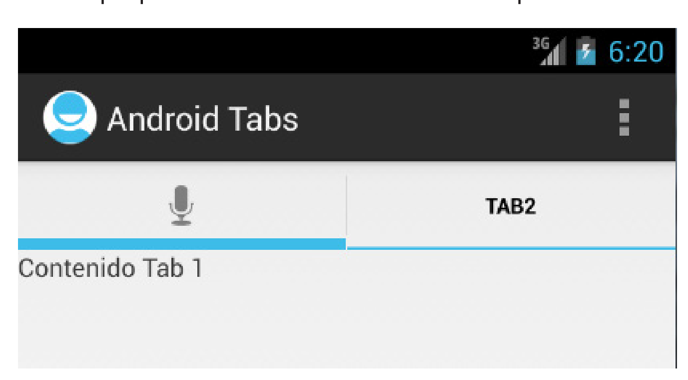




Una pequeña sorpresa. Como podéis comprobar, aunque estamos indicando en todos los casos un texto y un icono para cada pestaña, el comportamiento difiere entre las distintas versiones de Android. En Android 4, el comportamiento por defecto del control TabHost es mostrar sólo el texto, o solo el icono, pero no ambos. Si eliminamos el texto de la primera pestaña y volvemos a ejecutar veremos como el icono sí aparece.

```
TabHost.TabSpec spec=tabs.newTabSpec("mitab1");
        spec.setContent(R.id.tab1);
        spec.setIndicator("",
                  res.getDrawable(android.R.drawable.ic_btn_speak_now));
        tabs.addTab(spec);
```

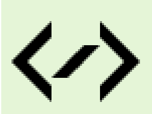
Con la pequeña modificación anterior la aplicación se vería así:



En cuanto a los eventos disponibles del control `TabHost`, aunque no suele ser necesario capturarlos, podemos ver a modo de ejemplo el más interesante de ellos, `OnTabChanged`, que se lanza cada vez que se cambia de pestaña y que nos informa de la nueva pestaña visualizada. Este evento podemos implementarlo y asignarlo mediante el método `setOnTabChangedListener()` de la siguiente forma:

```
tabs.setOnTabChangedListener(new OnTabChangeListener() {
    @Override
    public void onTabChanged(String tabId) {
        Log.i("AndroidTabsDemo", "Pulsada pestaña: " + tabId);
    }
});
```

En el método `onTabChanged()` recibimos como parámetro la etiqueta identificativa de la pestaña (no su ID), que debimos asignar cuando creamos su objeto `TabSpec` correspondiente. Para este ejemplo, lo único que haremos al detectar un cambio de pestaña será escribir en el log de la aplicación un mensaje informativo con la etiqueta de la nueva pestaña visualizada. Así por ejemplo, al cambiar a la segunda pestaña recibiremos el mensaje de log: "*Pulsada pestaña: mitab2*".



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-tabs

Controles personalizados: Extender controles

En apartados anteriores hemos conocido y aprendido a utilizar muchos de los controles que proporciona Android en su SDK. Con la ayuda de estos controles podemos diseñar interfaces gráficas de lo más variopinto pero en ocasiones, si queremos dar un toque especial y original a nuestra aplicación, o simplemente si necesitamos cierta funcionalidad no presente en los componentes estándar de Android, nos vemos en la necesidad de crear nuestros propios controles personalizados, diseñados a la medida de nuestros requisitos.

Android admite por supuesto crear controles personalizados, y permite hacerlo de diferentes formas:

1. Extendiendo la funcionalidad de un control ya existente.
2. Combinando varios controles para formar otro más complejo.
3. Diseñando desde cero un nuevo control.

En este primer apartado sobre el tema vamos a hablar de la primera opción, es decir, vamos a ver cómo podemos crear un nuevo control partiendo de la base de un control ya existente. A modo de ejemplo, vamos a extender el control `EditText` (cuadro de texto) para que muestre en todo momento el número de caracteres que contiene a medida que se escribe en él. Intentaríamos emular algo así como el editor de mensajes SMS del propio sistema operativo, que nos avisa del número de caracteres que contiene el mensaje.

En la esquina superior derecha del cuadro de texto vamos a mostrar el número de caracteres del mensaje de texto introducido, que irá actualizándose a medida que modificamos el texto. Para empezar, vamos a crear una nueva clase java que extienda del control que queremos utilizar como base, en este caso `EditText`.

```
public class ExtendedEditText extends EditText
{
    //...
}
```

Tras esto, sobrescribiremos siempre los tres constructores heredados, donde por el momento nos limitaremos a llamar al mismo constructor de la clase padre.

```
public ExtendedEditText(Context context, AttributeSet attrs, int defStyle){
    super(context, attrs,defStyle);
}

public ExtendedEditText(Context context, AttributeSet attrs) {
    super(context, attrs);
}

public ExtendedEditText(Context context) {
    super(context);
}
```

Por último el paso más importante. Dado que queremos modificar el aspecto del control para añadir el contador de caracteres tendremos que sobrescribir el evento `onDraw()`, que es llamado por Android cada vez que hay que redibujar el control en pantalla. Este método recibe como parámetro un objeto `Canvas`, que no es más que el "lienzo" sobre el que podemos dibujar todos los elementos extra necesarios en el control. El objeto `Canvas`, proporciona una serie de métodos para dibujar cualquier tipo de elemento (líneas, rectángulos, elipses, texto, bitmaps, ...) sobre el espacio ocupado por el control. En nuestro caso tan sólo vamos a necesitar dibujar sobre el control un rectángulo que sirva de fondo para el contador y el texto del contador con el número de caracteres actual del cuadro de texto. No vamos a entrar en muchos detalles sobre la forma de dibujar gráficos ya que ése será tema de otro apartado, pero vamos a ver al menos las acciones principales.

En primer lugar definiremos los "pinceles" (objetos `Paint`) que utilizaremos para dibujar, uno de ellos (`p1`) de color negro y relleno sólido para el fondo del contador, y otro (`p2`) de color blanco para el texto. Para configurar los colores, el estilo de fondo y el tamaño del texto utilizaremos los métodos `setColor()`, `setStyle()` y `setTextSize()` respectivamente:

```
private void inicializacion()
{
    Paint p1 = new Paint(Paint.ANTI_ALIAS_FLAG);
    p1.setColor(Color.BLACK);
    p1.setStyle(Style.FILL);

    Paint p2 = new Paint(Paint.ANTI_ALIAS_FLAG);
    p2.setColor(Color.WHITE);
    p2.setTextSize(20);
}
```

Dado que estos elementos tan sólo hará falta crearlos la primera vez que se dibuje el control, para evitar trabajo innecesario no incluiremos su definición en el método `onDraw()`, sino que los definiremos como atributos de la clase y los inicializaremos en el constructor del control (en los tres).

Una vez definidos los diferentes pinceles necesarios, dibujaremos el fondo y el texto del contador mediante los métodos `drawRect()` y `drawText()`, respectivamente, del objeto `canvas` recibido en el evento. Lo único a tener en cuenta es que todos estos métodos de dibujo reciben las unidades en pixeles y por tanto si utilizamos valores fijos tendremos problemas al visualizar los resultados en pantallas con distintas densidades de píxeles.

Para evitar esto en lo posible, tendremos que convertir nuestros valores de píxeles a algún valor dependiente de la densidad de la pantalla, lo que en Android podemos conseguir multiplicando siempre nuestros píxeles por un factor de escala que podemos obtener mediante los métodos `getResources().getDisplayMetrics().density`. Tras obtener este valor, multiplicaremos por él todas nuestras unidades en píxeles para conseguir los mismos efectos en cualquier pantalla. Veamos cómo quedaría el

código completo:

```
private void inicializacion()
{
      //...

      escala = getResources().getDisplayMetrics().density;
}

//...

@Override
public void onDraw(Canvas canvas)
{
    //Llamamos al método de la clase base (EditText)
    super.onDraw(canvas);

    //Dibujamos el fondo negro del contador
    canvas.drawRect(this.getWidth()-30*escala, 5*escala,
        this.getWidth()-5*escala, 20*escala, p1);

    //Dibujamos el número de caracteres sobre el contador
    canvas.drawText("" + this.getText().toString().length(),
        this.getWidth()-28*escala, 17*escala, p2);
}
```

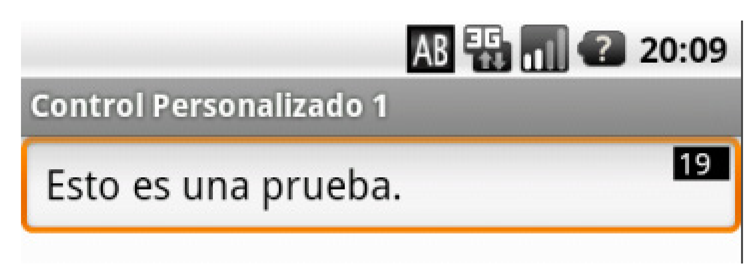
Como puede comprobarse, a estos métodos se les pasan como parámetro las coordenadas del elemento a dibujar relativas al espacio ocupado por el control y el pincel a utilizar en cada caso.

Hecho esto, ya tenemos finalizado nuestro cuadro de texto personalizado con contador de caracteres. Para añadirlo a la interfaz de nuestra aplicación lo incluiremos en el layout XML de la ventana tal como haríamos con cualquier otro control, teniendo en cuenta que deberemos hacer referencia a él con el nombre completo de la nueva clase creada (incluido el paquete java), que en mi caso particular sería `net.sgoliver.ExtendedEditText`.

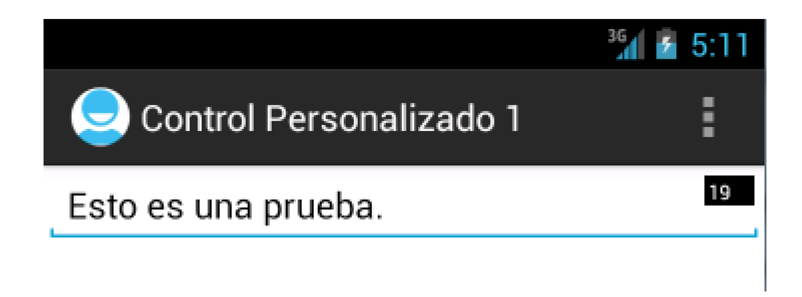
```
<net.sgoliver.android.controlpers1.ExtendedEditText
    android:id="@+id/TxtPrueba"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

Para finalizar, veamos cómo quedaría nuestro control en dos dispositivos distintos con densidades de pantalla diferentes.

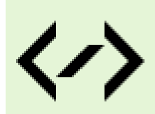
En Android 2 con densidad de pantalla baja:



Y en Android 4 con densidad de pantalla alta:



En el siguiente apartado veremos cómo crear un control personalizado utilizando la segunda de las opciones expuestas, es decir, combinando varios controles ya existentes. Comentaremos además como añadir eventos y propiedades personalizadas a nuestro control y cómo hacer referencia a dichas propiedades desde su definición XML.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

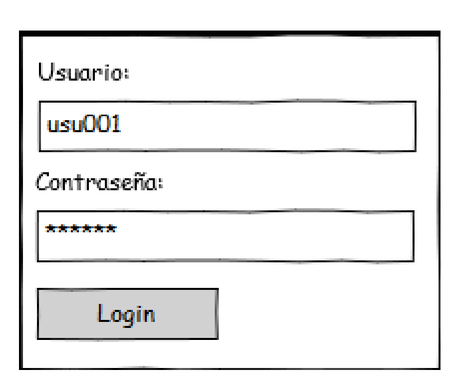
curso-android-src/android-control-pers-1

Controles personalizados: Combinar controles

Ya vimos cómo Android ofrece tres formas diferentes de crear controles personalizados para nuestras aplicaciones y dedicamos el capítulo anterior a comentar la primera de las posibilidades, que consistía en extender la funcionalidad de un control ya existente.

En este capítulo sobre el tema vamos a centrarnos en la creación de controles compuestos, es decir, controles personalizados construidos a partir de varios controles estándar, combinando la funcionalidad de todos ellos en un sólo control reutilizable en otras aplicaciones.

Como ejemplo ilustrativo vamos a crear un control de identificación (*login*) formado por varios controles estándar de Android. La idea es conseguir un control como el que se muestra la siguiente imagen esquemática:



A efectos didácticos, y para no complicar más el ejemplo, vamos a añadir también a la derecha del botón `Login` una etiqueta donde mostrar el resultado de la identificación del usuario (login correcto o incorrecto).

A este control añadiremos además eventos personalizados, veremos cómo añadirlo a nuestras aplicaciones, y haremos que se pueda personalizar su aspecto desde el layout XML de nuestra interfaz utilizando también atributos XML personalizados.

Empecemos por el principio. Lo primero que vamos a hacer es construir la interfaz de nuestro control a partir de controles sencillos: etiquetas, cuadros de texto y botones. Para ello vamos a crear un nuevo layout XML en la carpeta `\res\layout` con el nombre "`control_login.xml`".

En este fichero vamos a definir la estructura del control como ya hemos visto en muchos apartados anteriores,

sin ninguna particularidad destacable. Para este caso quedaría como sigue:

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical"
  android:padding="10dip">

      <TextView android:id="@+id/TextView01"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/usuario"
            android:textStyle="bold" />

      <EditText android:id="@+id/TxtUsuario"
            android:layout_height="wrap_content"
            android:layout_width="match_parent"
            android:inputType="text" />

      <TextView android:id="@+id/TextView02"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/contrasena"
            android:textStyle="bold" />

      <EditText android:id="@+id/TxtPassword"
            android:layout_height="wrap_content"
            android:layout_width="match_parent"
            android:inputType="textPassword" />

      <LinearLayout android:orientation="horizontal"
            android:layout_width="match_parent"
            android:layout_height="match_parent" >

            <Button
                  android:layout_width="wrap_content"
                  android:layout_height="wrap_content"
                  android:id="@+id/BtnAceptar"
                  android:text="@string/login"
                  android:paddingLeft="20dip"
                  android:paddingRight="20dip" />

            <TextView android:id="@+id/LblMensaje"
                  android:layout_width="wrap_content"
                  android:layout_height="wrap_content"
                  android:paddingLeft="10dip"
                  android:textStyle="bold" />
      </LinearLayout>

</LinearLayout>
```

A continuación crearemos su clase java asociada donde definiremos toda la funcionalidad de nuestro control. Dado que nos hemos basado en un `LinearLayout` para construir el control, esta nueva clase deberá heredar también de la clase java `LinearLayout` de Android. Redefiniremos además los dos constructores básicos:

```
package net.sgoliver.android.controlpers2;

//...

public class ControlLogin extends LinearLayout
{
public ControlLogin(Context context) {
    super(context);
    inicializar();
}

public ControlLogin(Context context, AttributeSet attrs) {
    super(context, attrs);
    inicializar();
}
}
```

Como se puede observar, todo el trabajo lo dejamos para el método `inicializar()`. En este método inflaremos el layout XML que hemos definido, obtendremos las referencias a todos los controles y asignaremos los eventos necesarios. Todo esto ya lo hemos hecho en otras ocasiones, por lo que tampoco nos vamos a detener mucho. Veamos cómo queda el método completo:

```
private void inicializar()
{
    //Utilizamos el layout 'control_login' como interfaz del control
    String infService = Context.LAYOUT_INFLATER_SERVICE;
    LayoutInflater li =
        (LayoutInflater)getContext().getSystemService(infService);
    li.inflate(R.layout.control_login, this, true);

    //Obtenemoslas referencias a los distintos control
    txtUsuario = (EditText)findViewById(R.id.TxtUsuario);
    txtPassword = (EditText)findViewById(R.id.TxtPassword);
    btnLogin = (Button)findViewById(R.id.BtnAceptar);
    lblMensaje = (TextView)findViewById(R.id.LblMensaje);

    //Asociamos los eventos necesarios
    asignarEventos();
}
```

Dejaremos por ahora a un lado el método `asignarEventos()`, volveremos sobre él más tarde.

Con esto ya tenemos definida la interfaz y la funcionalidad básica del nuevo control por lo que ya podemos utilizarlo desde otra actividad como si se tratase de cualquier otro control predefinido. Para ello haremos referencia a él utilizando la ruta completa del paquete java utilizado, en nuestro caso quedaría como `net.sgoliver.ControlLogin`. Vamos a insertar nuestro nuevo control en la actividad principal de la aplicación:

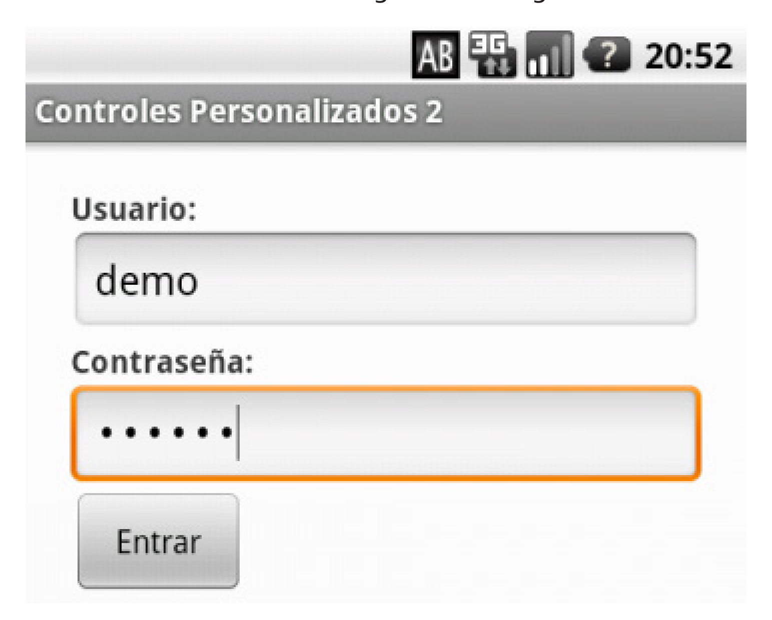
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:sgo="http://schemas.android.com/apk/res/net.sgoliver.android.
controlpers2"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="10dip"  >
```

```
    <net.sgoliver.android.controlpers2.ControlLogin
        android:id="@+id/CtlLogin"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

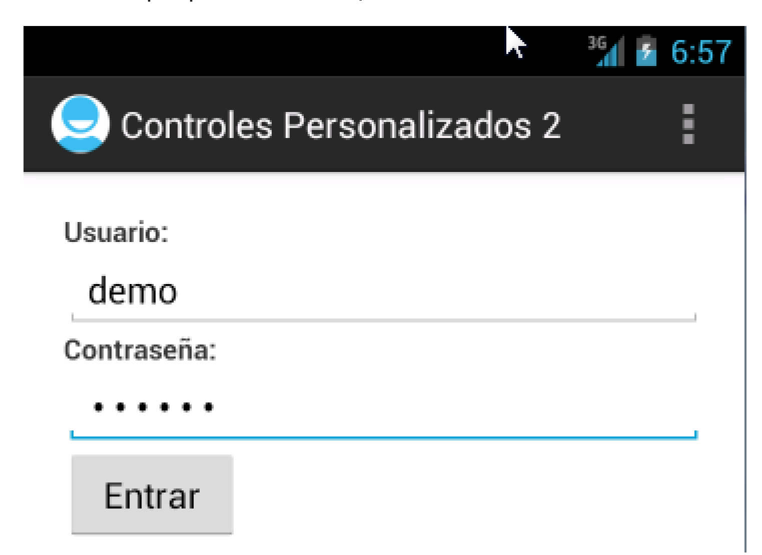
</LinearLayout>
```

Dado que estamos heredando de un `LinearLayout` podemos utilizar en principio cualquier atributo permitido para dicho tipo de controles, en este caso hemos establecido por ejemplo los atributos `layout_width`, `layout_height` y `background`. Si ejecutamos ahora la aplicación veremos cómo ya hemos conseguido gran parte de nuestro objetivo.

En Android 2.x tendríamos algo como lo siguiente:



Mientras que para Android 4.x lo veríamos más o menos así:



Vamos a añadir ahora algo más de funcionalidad. En primer lugar, podemos añadir algún método público exclusivo de nuestro control. Como ejemplo podemos añadir un método que permita modificar el texto de la etiqueta de resultado del login.

```
public void setMensaje(String msg)
{
     lblMensaje.setText(msg);
}
```

En segundo lugar, todo control que se precie debe tener algunos eventos que nos permitan responder a las acciones del usuario de la aplicación. Así por ejemplo, los botones tienen un evento `OnClick`, las listas un evento `OnItemSelected`, etc. Pues bien, nosotros vamos a dotar a nuestro control de un evento personalizado, llamado `OnLogin`, que se lance cuando el usuario introduce sus credenciales de identificación y pulsa el botón "*Login*".

Para ello, lo primero que vamos a hacer es concretar los detalles de dicho evento, creando una interfaz java para definir su *listener*. Esta interfaz tan sólo tendrá un método llamado `onLogin()` que devolverá los dos datos introducidos por el usuario (usuario y contraseña).

```
package net.sgoliver.android.controlpers2;

public interface OnLoginListener {
      void onLogin(String usuario, String password);
}
```

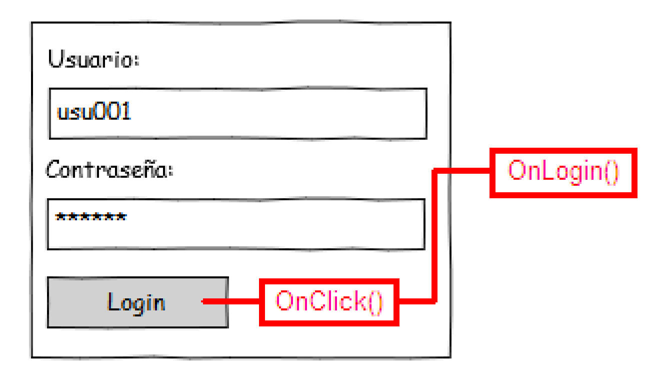
A continuación, deberemos añadir un nuevo miembro de tipo `OnLoginListener` a la clase `ControlLogin`, y un método público que permita suscribirse al nuevo evento.

```
public class ControlLogin extends LinearLayout
{
     private OnLoginListener listener;

     //...

     public void setOnLoginListener(OnLoginListener l)
     {
      listener = l;
     }
}
```

Con esto, la aplicación principal ya puede suscribirse al evento `OnLogin` y ejecutar su propio código cuando éste se genere. ¿Pero cuándo se genera exactamente? Dijimos antes que queremos lanzar el evento `OnLogin` cuando el usuario pulse el botón "*Login*" de nuestro control. Pues bien, para hacerlo, volvamos al método `asignarEventos()` que antes dejamos aparcado. En este método vamos a implementar el evento `OnClick` del botón de *Login* para lanzar el nuevo evento `OnLogin` del control. ¿Confundido?. Intento explicarlo de otra forma. Vamos a aprovechar el evento `OnClick()` del botón *Login* (que es un evento interno a nuestro control, no se verá desde fuera) para lanzar hacia el exterior el evento `OnLogin()` (que será el que debe capturar y tratar la aplicación que haga uso del control).



Para ello, implementaremos el evento `OnClick` como ya hemos hecho en otras ocasiones y como acciones generaremos el evento `OnLogin` de nuestro *listener* pasándole los datos que ha introducido el usuario en los cuadros de texto "*Usuario*" y "*Contraseña*":

```
private void asignarEventos()
{
    btnLogin.setOnClickListener(new OnClickListener()
    {
        @Override
        public void onClick(View v) {
            listener.onLogin(txtUsuario.getText().toString(),
                             txtPassword.getText().toString());
        }
    });
}
```

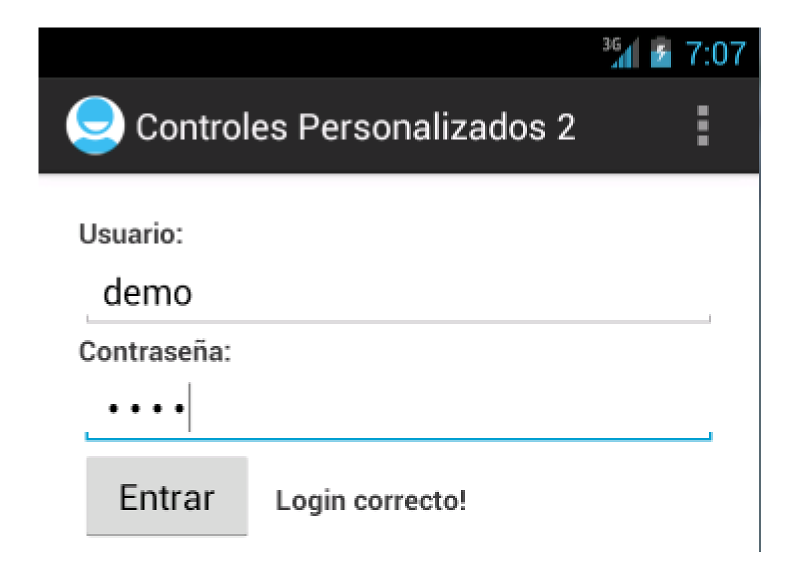
Con todo esto, la aplicación principal ya puede implementar el evento `OnLogin` de nuestro control, haciendo por ejemplo la validación de las credenciales del usuario y modificando convenientemente el texto de la etiqueta de resultado:

```
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ctlLogin = (ControlLogin)findViewById(R.id.CtlLogin);

    ctlLogin.setOnLoginListener(new OnLoginListener()
    {
        @Override
        public void onLogin(String usuario, String password)
        {
            //Validamos el usuario y la contraseña
            if (usuario.equals("demo") && password.equals("demo"))
                ctlLogin.setMensaje("Login correcto!");
            else
                ctlLogin.setMensaje("Vuelva a intentarlo.");
        }
    });
}
```

Veamos lo que ocurre al ejecutar ahora la aplicación principal e introducir las credenciales correctas (por ejemplo para Android 4):



Nuestro control está ya completo, en aspecto y funcionalidad. Hemos personalizado su interfaz y hemos añadido métodos y eventos propios. ¿Podemos hacer algo más? Pues sí.

Cuando vimos cómo añadir el control de login al layout de la aplicación principal dijimos que podíamos utilizar cualquier atributo XML permitido para el contenedor `LinearLayout`, ya que nuestro control derivaba de éste. Pero vamos a ir más allá y vamos a definir también atributos XML exclusivos para nuestro control. Como ejemplo, vamos a definir un atributo llamado `login_text` que permita establecer el texto del botón de Login desde el propio layout XML, es decir, en tiempo de diseño.

Primero vamos de declarar el nuevo atributo y lo vamos a asociar al control `ControlLogin`. Esto debe hacerse en el fichero `\res\values\attrs.xml`. Para ello, añadiremos una nueva sección `<declare-styleable>` asociada a `ControlLogin` dentro del elemento `<resources>`, donde indicaremos el nombre (`name`) y el tipo (`format`) del nuevo atributo.

```
<resources>
    <declare-styleable name="ControlLogin">
        <attr name="login_text" format="string"/>
    </declare-styleable>
</resources>
```

En nuestro caso, el tipo del atributo será `string`, dado que contendrá una cadena de texto con el mensaje a mostrar en el botón.

Con esto ya tendremos permitido el uso del nuevo atributo dentro del layout de la aplicación principal. Ahora nos falta procesar el atributo desde nuestro control personalizado. Este tratamiento lo podemos hacer en el constructor de la clase `ControlLogin`. Para ello, obtendremos la lista de atributos asociados a `ControlLogin` mediante el método `obtainStyledAttributes()` del contexto de la aplicación, obtendremos el valor del nuevo atributo definido (mediante su ID, que estará formado por la concatenación del nombre del control y el nombre del atributo, en nuestro caso `ControlLogin_login_text`) y modificaremos el texto por defecto del control con el nuevo texto.

```
public ControlLogin(Context context, AttributeSet attrs) {
    super(context, attrs);
    inicializar();

    // Procesamos los atributos XML personalizados
    TypedArray a =
        getContext().obtainStyledAttributes(attrs,
            R.styleable.ControlLogin);
```

```
    String textoBoton = a.getString(
            R.styleable.ControlLogin_login_text);

    btnLogin.setText(textoBoton);

    a.recycle();
}
```

Ya sólo nos queda utilizarlo. Para ello debemos primero declarar un nuevo espacio de nombres (*namespace*) local para el paquete java utilizado, que en nuestro caso he llamado "sgo":

```
xmlns:sgo="http://schemas.android.com/apk/res/net.sgoliver.android.
controlpers2"
```

Tras esto, sólo queda asignar el valor del nuevo atributo precedido del nuevo namespace, por ejemplo con el texto "Entrar":

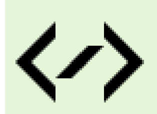
```
<LinearLayout
xmlns:android=http://schemas.android.com/apk/res/android xmlns:sgo="http://
schemas.android.com/apk/res/net.sgoliver.android.controlpers2"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="10dip" >

    <net.sgoliver.android.controlpers2.ControlLogin
        android:id="@+id/CtlLogin"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        sgo:login_text="Entrar" />

</LinearLayout>
```

Con esto, conseguiríamos el mismo efecto que los ejemplos antes mostrados, pero de una forma mucho más fácilmente personalizable, con el texto del botón controlado directamente por un atributo del layout XML.

Como resumen, en este apartado hemos visto cómo construir un control Android personalizado a partir de otros controles estándar, componiendo su interfaz, añadiendo métodos y eventos personalizados, e incluso añadiendo nuevas opciones en tiempo de diseño añadiendo atributos XML exclusivos.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-control-pers-2

Controles personalizados: Diseño completo

En apartados anteriores del curso ya comentamos dos de las posibles vías que tenemos para crear controles personalizados en Android: la primera de ellas extendiendo la funcionalidad de un control ya existente, y como segunda opción creando un nuevo control compuesto por otros más sencillos.

En este nuevo apartado vamos a describir la tercera de las posibilidades que teníamos disponibles, que consiste en crear un control completamente desde cero, sin utilizar como base otros controles existentes.

Como ejemplo, vamos a construir un control que nos permita seleccionar un color entre varios disponibles.

Los colores disponibles van a ser sólo cuatro, que se mostrarán en la franja superior del control. En la parte inferior se mostrará el color seleccionado en cada momento, o permanecerá negro si aún no se ha seleccionado ningún color. Valga la siguiente imagen como muestra del aspecto que tendrá nuestro control de selección de color en Android 2:



En Android 4 se visualizaría de forma prácticamente idéntica:



Por supuesto este control no tiene mucha utilidad práctica dada su sencillez, pero creo que puede servir como ejemplo para comentar los pasos necesarios para construir cualquier otro control más complejo. Empecemos.

En las anteriores ocasiones vimos cómo el nuevo control creado siempre heredaba de algún otro control o contenedor ya existente. En este caso sin embargo, vamos a heredar nuestro control directamente de la clase `View` (clase padre de la gran mayoría de elementos visuales de Android). Esto implica, entre otras cosas, que por defecto nuestro control no va a tener ningún tipo de interfaz gráfica, por lo que todo el trabajo de "dibujar" la interfaz lo vamos a tener que hacer nosotros. Además, como paso previo a la representación gráfica de la interfaz, también vamos a tener que determinar las dimensiones que nuestro control tendrá dentro de su elemento contenedor. Como veremos ahora, ambas cosas se llevarán a cabo redefiniendo dos eventos de la clase View, `onDraw()` para el dibujo de la interfaz, y `onMeasure()` para el cálculo de las dimensiones.

Por llevar un orden cronológico, empecemos comentando el evento `onMeasure()`. Este evento se ejecuta automáticamente cada vez que se necesita recalcular el tamaño de un control. Pero como ya hemos visto en varias ocasiones, los elementos gráficos incluidos en una aplicación Android se distribuyen por la pantalla de una forma u otra dependiendo del tipo de contenedor o layout utilizado. Por tanto, el tamaño de un control determinado en la pantalla no dependerá sólo de él, sino de ciertas restricciones impuestas por su elemento contenedor o elemento padre. Para resolver esto, en el evento `onMeasure()` recibiremos como parámetros las restricciones del elemento padre en cuanto a ancho y alto del control, con lo que podremos tenerlas en cuenta a la hora de determinar el ancho y alto de nuestro control personalizado. Estas restricciones se reciben en forma de objetos `MeasureSpec`, que contiene dos campos: modo y tamaño. El significado del segundo de ellos es obvio, el primero por su parte sirve para matizar el significado del

segundo. Me explico. Este campo modo puede contener tres valores posibles:

Valor	Descripción
AT_MOST	Indica que el control podrá tener como máximo el tamaño especificado.
EXACTLY	Indica que al control se le dará exactamente el tamaño especificado.
UNSPECIFIED	Indica que el control padre no impone ninguna restricción sobre el tamaño.

Dependiendo de esta pareja de datos, podremos calcular el tamaño deseado para nuestro control. Para nuestro control de ejemplo, apuraremos siempre el tamaño máximo disponible (o un tamaño por defecto de 200*100 en caso de no recibir ninguna restricción), por lo que en todos los casos elegiremos como tamaño de nuestro control el tamaño recibido como parámetro:

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec)
{
    int ancho = calcularAncho(widthMeasureSpec);
    int alto = calcularAlto(heightMeasureSpec);

    setMeasuredDimension(ancho, alto);
}

private int calcularAlto(int limitesSpec)
{
    int res = 100; //Alto por defecto

    int modo = MeasureSpec.getMode(limitesSpec);
    int limite = MeasureSpec.getSize(limitesSpec);

    if (modo == MeasureSpec.AT_MOST) {
        res = limite;
    }
    else if (modo == MeasureSpec.EXACTLY) {
        res = limite;
    }

    return res;
}

private int calcularAncho(int limitesSpec)
{
    int res = 200; //Ancho por defecto

    int modo = MeasureSpec.getMode(limitesSpec);
    int limite = MeasureSpec.getSize(limitesSpec);

    if (modo == MeasureSpec.AT_MOST) {
        res = limite;
    }
    else if (modo == MeasureSpec.EXACTLY) {
        res = limite;
    }

    return res;
}
```

Como nota importante, al final del evento onMeasure() siempre debemos llamar al método setMeasuredDimension() pasando como parámetros el ancho y alto calculados para nuestro control.

Con esto ya hemos determinado las dimensiones del control, por lo que tan sólo nos queda dibujar su interfaz gráfica. Como hemos indicado antes, esta tarea se realiza dentro del evento `onDraw()`. Este evento recibe como parámetro un objeto de tipo `Canvas`, sobre el que podremos ejecutar todas las operaciones de dibujo de la interfaz. No voy a entrar en detalles de la clase `Canvas`, porque ése será tema central de un próximo apartado. Por ahora nos vamos a conformar sabiendo que es la clase que contiene la mayor parte de los métodos de dibujo en interfaces Android, por ejemplo `drawRect()` para dibujar rectángulos, `drawCircle()` para círculos, `drawBitmap()` para imagenes, `drawText()` para texto, e infinidad de posibilidades más. Para consultar todos los métodos disponibles puedes dirigirte a la documentación oficial de la clase `Canvas` de Android. Además de la clase `Canvas`, también me gustaría destacar la clase `Paint`, que permite definir el estilo de dibujo a utilizar en los métodos de dibujo de `Canvas`, por ejemplo el ancho de trazado de las líneas, los colores de relleno, etc.

Para nuestro ejemplo no necesitaríamos conocer nada más, ya que la interfaz del control es relativamente sencilla. Vemos primero el código y después comentamos los pasos realizados:

```
@Override
protected void onDraw(Canvas canvas)
{
    //Obtenemos las dimensiones del control
    int alto = getMeasuredHeight();
    int ancho = getMeasuredWidth();

    //Colores Disponibles
    Paint pRelleno = new Paint();
    pRelleno.setStyle(Style.FILL);

    pRelleno.setColor(Color.RED);
    canvas.drawRect(0, 0, ancho/4, alto/2, pRelleno);

    pRelleno.setColor(Color.GREEN);
    canvas.drawRect(ancho/4, 0, 2*(ancho/4), alto/2, pRelleno);

    pRelleno.setColor(Color.BLUE);
    canvas.drawRect(2*(ancho/4), 0, 3*(ancho/4), alto/2, pRelleno);

    pRelleno.setColor(Color.YELLOW);
    canvas.drawRect(3*(ancho/4), 0, 4*(ancho/4), alto/2, pRelleno);

    //Color Seleccionado
    pRelleno.setColor(colorSeleccionado);
    canvas.drawRect(0, alto/2, ancho, alto, pRelleno);

    //Marco del control
    Paint pBorde = new Paint();
    pBorde.setStyle(Style.STROKE);
    pBorde.setColor(Color.WHITE);
    canvas.drawRect(0, 0, ancho-1, alto/2, pBorde);
    canvas.drawRect(0, 0, ancho-1, alto-1, pBorde);
}
```

En primer lugar obtenemos las dimensiones calculadas en la última llamada a `onMeasure()` mediante los métodos `getMeasuredHeight()` y `getMeasuredWidth()`. Posteriormente definimos un objeto `Paint` que usaremos para dibujar los rellenos de cada color seleccionable. Para indicar que se trata del color de relleno a utilizar utilizaremos la llamada a `setStyle(Style.FILL)`. Tras esto, ya sólo debemos dibujar cada uno de los cuadros en su posición correspondiente con `drawRect()`, estableciendo antes de cada uno de ellos el color deseado con `setColor()`. Por último, dibujamos el marco del control definiendo un nuevo objeto `Paint`, esta vez con estilo `Style.STROKE` dado que se utilizará para dibujar sólo líneas, no rellenos.

Con esto, ya deberíamos tener un control con el aspecto exacto que definimos en un principio. El siguiente paso será definir su funcionalidad implementando los eventos a los que queramos que responda nuestro control, tanto eventos internos como externos.

En nuestro caso sólo vamos a tener un evento de cada tipo. En primer lugar definiremos un evento interno (evento que sólo queremos capturar de forma interna al control, sin exponerlo al usuario) para responder a las pulsaciones del usuario sobre los colores de la zona superior, y que utilizaremos para actualizar el color de la zona inferior con el color seleccionado. Para ello implementaremos el evento `onTouch()`, lanzado cada vez que el usuario toca la pantalla sobre nuestro control. La lógica será sencilla, simplemente consultaremos las coordenadas donde ha pulsado el usuario (mediante los métodos `getX()` y `getY()`), y dependiendo del lugar pulsado determinaremos el color sobre el que se ha seleccionado y actualizaremos el valor del atributo `colorSeleccionado`. Finalmente, llamamos al método `invalidate()` para refrescar la interfaz del control, reflejando así el cambio en el color seleccionado, si se ha producido.

```
@Override
public boolean onTouchEvent(MotionEvent event)
{
    //Si se ha pulsado en la zona superior
    if (event.getY() > 0 && event.getY() < (getMeasuredHeight()/2))
    {
        //Si se ha pulsado dentro de los límites del control
        if (event.getX() > 0 && event.getX() < getMeasuredWidth())
        {
            //Determinamos el color seleccionado según el punto pulsado
            if(event.getX() > (getMeasuredWidth()/4)*3)
                colorSeleccionado = Color.YELLOW;
            else if(event.getX() > (getMeasuredWidth()/4)*2)
                colorSeleccionado = Color.BLUE;
            else if(event.getX() > (getMeasuredWidth()/4)*1)
                colorSeleccionado = Color.GREEN;
            else
                colorSeleccionado = Color.RED;

            //Refrescamos el control
            this.invalidate();
        }
    }

    return super.onTouchEvent(event);
}
```

En segundo lugar crearemos un evento externo personalizado, que lanzaremos cuando el usuario pulse sobre la zona inferior del control, como una forma de aceptar definitivamente el color seleccionado. Llamaremos a este evento `onSelectedColor()`. Para crearlo actuaremos de la misma forma que ya vimos en el capítulo anterior. Primero definiremos una interfaz para el listener de nuestro evento:

```
package net.sgoliver.android.controlpers3;

public interface OnColorSelectedListener
{
    void onColorSelected(int color);
}
```

Posteriormente, definiremos un objeto de este tipo como atributo de nuestro control y escribiremos un nuevo método que permita a las aplicaciones suscribirse al evento:

```
public class SelectorColor extends View
{
     private OnColorSelectedListener listener;

     //...

     public void setOnColorSelectedListener(OnColorSelectedListener l)
     {
         listener = l;
     }
}
```

Y ya sólo nos quedaría lanzar el evento en el momento preciso. Esto también lo haremos dentro del evento `onTouch()`, cuando detectemos que el usuario ha pulsado en la zona inferior de nuestro control:

```
@Override
public boolean onTouchEvent(MotionEvent event)
{
    //Si se ha pulsado en la zona superior
    if (event.getY() > 0 && event.getY() < (getMeasuredHeight()/2))
    {
        //...
    }
    //Si se ha pulsado en la zona inferior
    else if (event.getY() > (getMeasuredHeight()/2) &&
                 event.getY() < getMeasuredHeight())
    {
        //Lanzamos el evento externo de selección de color
        listener.onColorSelected(colorSeleccionado);
    }

    return super.onTouchEvent(event);
}
```

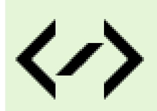
Con esto, nuestra aplicación principal ya podría suscribirse a este nuevo evento para estar informada cada vez que se seleccione un color. Sirva la siguiente plantilla a modo de ejemplo:

```
public class ControlPersonalizado extends Activity
{
    private SelectorColor ctlColor;

    @Override
    public void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        ctlColor = (SelectorColor)findViewById(R.id.scColor);
        ctlColor.setOnColorSelectedListener(new OnColorSelectedListener()
        {
            @Override
            public void onColorSelected(int color) {
                //Aquí se trataría el color seleccionado (parámetro 'color'
                //...
            }
        });
    }
}
```

Con esto, tendríamos finalizado nuestro control completamente personalizado, que hemos construido sin utilizar como base ningún otro control predefinido, definiendo desde cero tanto su aspecto visual como su funcionalidad interna o sus eventos públicos.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-control-pers-3

Fragments

Cuando empezaron a aparecer dispositivos de gran tamaño tipo tablet, el equipo de Android tuvo que solucionar el problema de la adaptación de la interfaz gráfica de las aplicaciones a ese nuevo tipo de pantallas. Una interfaz de usuario diseñada para un teléfono móvil no se adaptaba fácilmente a una pantalla 4 o 5 pulgadas mayor. La solución a esto vino en forma de un nuevo tipo de componente llamado `Fragment`.

Un *fragment* no puede considerarse ni un control ni un contenedor, aunque se parecería más a lo segundo. Un fragment podría definirse como una porción de la interfaz de usuario que puede añadirse o eliminarse de una interfaz de forma independiente al resto de elementos de la actividad, y que por supuesto puede reutilizarse en otras actividades. Esto, aunque en principio puede parecer algo trivial, nos va a permitir poder dividir nuestra interfaz en varias porciones de forma que podamos diseñar diversas configuraciones de pantalla, dependiendo de su tamaño y orientación, sin tener que duplicar código en ningún momento, sino tan sólo utilizando o no los distintos fragmentos para cada una de las posibles configuraciones. Intentemos aclarar esto un poco con un ejemplo.

No quería utilizar el ejemplo típico que aparece por todos lados, pero en este caso creo que es el más ilustrativo. Supongamos una aplicación de correo electrónico, en la que por un lado debemos mostrar la lista de correos disponibles, con sus campos clásicos *De* y *Asunto*, y por otro lado debemos mostrar el contenido completo del correo seleccionado. En un teléfono móvil lo habitual será tener una primera actividad que muestre el listado de correos, y cuando el usuario seleccione uno de ellos navegar a una nueva actividad que muestre el contenido de dicho correo. Sin embargo, en un tablet puede existir espacio suficiente para tener ambas partes de la interfaz en la misma pantalla, por ejemplo en un tablet en posición horizontal podríamos tener una columna a la izquierda con el listado de correos y dedicar la zona derecha a mostrar el detalle del correo seleccionado, todo ello sin tener que cambiar de actividad.

Antes de existir los fragments podríamos haber hecho esto implementando diferentes actividades con diferentes layouts para cada configuración de pantalla, pero esto nos habría obligado a duplicar gran parte del código en cada actividad. Tras la aparición de los fragments, colocaríamos el listado de correos en un fragment y la vista de detalle en otro, cada uno de ellos acompañado de su lógica de negocio asociada, y tan sólo nos quedaría definir varios layouts para cada configuración de pantalla incluyendo [o no] cada uno de estos fragments.

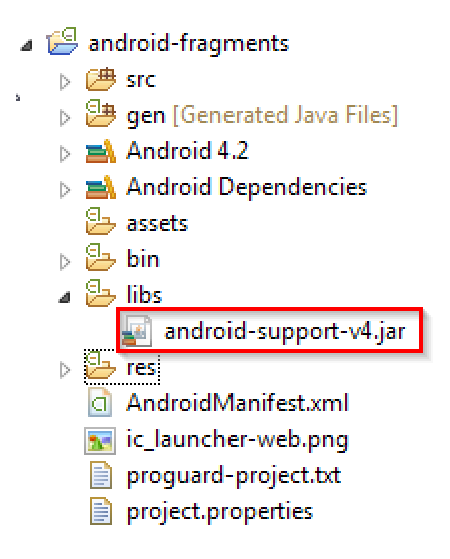
A modo de aplicación de ejemplo para este apartado, nosotros vamos a simular la aplicación de correo que hemos comentado antes, adaptándola a tres configuraciones distintas: pantalla normal, pantalla grande horizontal y pantalla grande vertical. Para el primer caso colocaremos el listado de correos en una actividad y el detalle en otra, mientras que para el segundo y el tercero ambos elementos estarán en la misma actividad, a derecha/izquierda para el caso horizontal, y arriba/abajo en el caso vertical.

Definiremos por tanto dos fragments: uno para el listado y otro para la vista de detalles. Ambos serán muy sencillos. Al igual que una actividad, cada fragment se compondrá de un fichero de layout XML para la interfaz (colocado en alguna carpeta `/res/layout`) y una clase java para la lógica asociada.

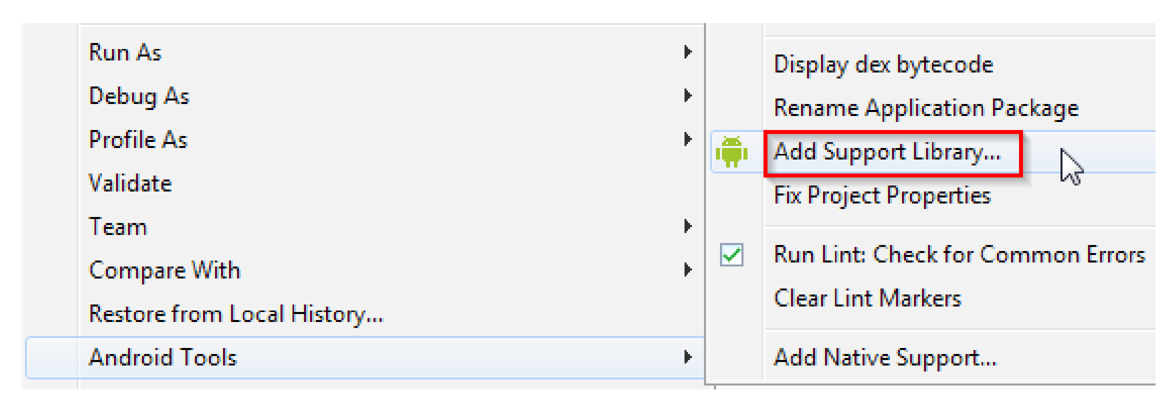
El primero de los fragment a definir contendrá tan sólo un control `ListView`, para el que definiremos un adaptador personalizado para mostrar dos campos por fila ("De" y "Asunto"). Ya describimos cómo hacer esto en el apartado dedicado al control `ListView`. El layout XML (lo llamaremos `fragment_listado.xml`) quedaría por tanto de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <ListView
        android:id="@+id/LstListado"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
    </ListView>
</LinearLayout>
```

Como hemos dicho, todo fragment debe tener asociada, además del layout, su propia clase java, que en este caso debe extender de la clase `Fragment`. Y aquí viene el primer contratiempo. Los fragment aparecieron con la versión 3 de Android, por lo que en principio no estarían disponibles para versiones anteriores. Sin embargo, Google pensó en todos y proporcionó esta característica también como parte de la librería de compatibillidad `android-support`, que en versiones recientes del plugin de Android para Eclipse se añade por defecto a todos los proyectos creados.



Si no fuera así, también puede incluirse manualmente en el proyecto mediante la opción "*Add Support Library...*" del menú contextual del proyecto.



Hecho esto, ya no habría ningún problema para utilizar la clase `Fragment`, y otras que comentaremos más adelante, para utilizar fragmentos compatibles con la mayoría de versiones de Android. Veamos cómo quedaría nuestra clase asociada al fragment de listado.

```
package net.sgoliver.android.fragments;

import android.app.Activity;
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;

public class FragmentListado extends Fragment {

    private Correo[] datos = new Correo[]{
        new Correo("Persona 1", "Asunto del correo 1", "Texto del correo 1"),
        new Correo("Persona 2", "Asunto del correo 2", "Texto del correo 2"),
        new Correo("Persona 3", "Asunto del correo 3", "Texto del correo 3"),
        new Correo("Persona 4", "Asunto del correo 4", "Texto del correo 4"),
        new Correo("Persona 5", "Asunto del correo 5", "Texto del correo 5")};

    private ListView lstListado;

    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState) {

        return inflater.inflate(R.layout.fragment_listado, container, false);
    }

    @Override
    public void onActivityCreated(Bundle state) {
        super.onActivityCreated(state);

        lstListado = (ListView)getView().findViewById(R.id.LstListado);

        lstListado.setAdapter(new AdaptadorCorreos(this));
    }

    class AdaptadorCorreos extends ArrayAdapter<Correo> {

            Activity context;

            AdaptadorCorreos(Fragment context) {
                super(context.getActivity(), R.layout.listitem_correo, datos);
                this.context = context.getActivity();
            }

            public View getView(int position, View convertView,
                                ViewGroup parent) {

                LayoutInflater inflater = context.getLayoutInflater();
                View item = inflater.inflate(R.layout.listitem_correo, null);

                TextView lblDe =
                            (TextView)item.findViewById(R.id.LblDe);
```

```
                lblDe.setText(datos[position].getDe());

                TextView lblAsunto =
                            (TextView)item.findViewById(R.id.LblAsunto);

                lblAsunto.setText(datos[position].getAsunto());

            return(item);
        }
    }
}
```

La clase `Correo` es una clase sencilla, que almacena los campos *De*, *Asunto* y *Texto* de un correo.

```
package net.sgoliver.android.fragments;

public class Correo
{
    private String de;
    private String asunto;
    private String texto;

    public Correo(String de, String asunto, String texto){
        this.de = de;
        this.asunto = asunto;
        this.texto = texto;
    }

    public String getDe(){
        return de;
    }

    public String getAsunto(){
        return asunto;
    }

    public String getTexto(){
        return texto;
    }
}
```

Si observamos con detenimiento las clases anteriores veremos que no existe casi ninguna diferencia con los temas ya comentados en apartados anteriores del curso sobre utilización de controles de tipo lista y adaptadores personalizados. La única diferencia que encontramos aquí respecto a ejemplos anteriores, donde definíamos actividades en vez de fragments, son los métodos que sobrescribimos. En el caso de los fragment son normalmente dos: `onCreateView()` y `onActivityCreated()`.

El primero de ellos, `onCreateView()`, es el equivalente al `onCreate()` de las actividades, es decir, que dentro de él es donde normalmente asignaremos un layout determinado al fragment. En este caso tendremos que "inflarlo" mediante el método `inflate()` pasándole como parámetro el ID del layout correspondiente, en nuestro caso `fragment_listado`.

El segundo de los métodos, `onActivityCreated()`, se ejecutará cuando la actividad contenedora del fragment esté completamente creada. En nuestro caso, estamos aprovechando este evento para obtener la referencia al control `ListView` y asociarle su adaptador. Sobre la definición del adaptador personalizado `AdaptadorCorreos` no comentaremos nada porque es idéntico al ya descrito en el apartado sobre listas.

Con esto ya tenemos creado nuestro fragment de listado, por lo que podemos pasar al segundo, que como ya dijimos se encargará de mostrar la vista de detalle. La definición de este fragment será aún más sencilla que la anterior. El layout, que llamaremos `fragment_detalle.xml`, tan sólo se compondrá de un cuadro de texto:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="#FFBBBBBB" >

    <TextView
        android:id="@+id/TxtDetalle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Y por su parte, la clase java asociada, se limitará a inflar el layout de la interfaz. Adicionalmente añadiremos un método público, llamado `mostrarDetalle()`, que nos ayude posteriormente a asignar el contenido a mostrar en el cuadro de texto.

```
package net.sgoliver.android.fragments;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class FragmentDetalle extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState) {

        return inflater.inflate(R.layout.fragment_detalle, container, false);
    }

    public void mostrarDetalle(String texto) {
        TextView txtDetalle =
            (TextView)getView().findViewById(R.id.TxtDetalle);

        txtDetalle.setText(texto);
    }
}
```

Una vez definidos los dos fragments, ya tan solo nos queda definir las actividades de nuestra aplicación, con sus respectivos layouts que harán uso de los fragments que acabamos de implementar.

Para la actividad principal definiremos 3 layouts diferentes: el primero de ellos para los casos en los que la aplicación se ejecute en una pantalla normal (por ejemplo un teléfono móvil) y dos para pantallas grandes (uno pensado para orientación horizontal y otro para vertical). Todos se llamarán `activity_main.xml`, y lo que marcará la diferencia será la carpeta en la que colocamos cada uno. Así, el primero de ellos

lo colocaremos en la carpeta por defecto /res/layout, y los otros dos en las carpetas /res/layout-large (pantalla grande) y /res/latout-large-port (pantalla grande con orientación vertical) respectivamente. De esta forma, según el tamaño y orientación de la pantalla Android utilizará un layout u otro de forma automática sin que nosotros tengamos que hacer nada.

Para el caso de pantalla normal, la actividad principal mostrará tan sólo el listado de correos, por lo que el layout incluirá tan sólo el fragment FragmentListado.

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    class="net.sgoliver.android.fragments.FragmentListado"
    android:id="@+id/FrgListado"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Como podéis ver, para incluir un fragment en un layout utilizaremos una etiqueta <fragment> con un atributo class que indique la ruta completa de la clase java correspondiente al fragment, en este primer caso "net.sgoliver.android.fragments.FragmentListado". Los demás atributos utilizados son los que ya conocemos de id, layout_width y layout_height.

En este caso de pantalla normal, la vista de detalle se mostrará en una segunda actividad, por lo que también tendremos que crear su layout, que llamaremos activity_detalle.xml. Veamos rápidamente su implementación:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    class="net.sgoliver.android.fragments.FragmentDetalle"
    android:id="@+id/FrgDetalle"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Como vemos es análoga a la anterior, con la única diferencia de que añadimos el fragment de detalle en vez de el de listado.

Por su parte, el layout para el caso de pantalla grande horizontal, será de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment class="net.sgoliver.android.fragments.FragmentListado"
        android:id="@+id/FrgListado"
        android:layout_weight="30"
        android:layout_width="0px"
        android:layout_height="match_parent" />

    <fragment class="net.sgoliver.android.fragments.FragmentDetalle"
        android:id="@+id/FrgDetalle"
        android:layout_weight="70"
        android:layout_width="0px"
        android:layout_height="match_parent" />

</LinearLayout>
```

Como veis en este caso incluimos los dos fragment en la misma pantalla, ya que tendremos espacio de sobra, ambos dentro de un `LinearLayout` horizontal, asignando al primero de ellos un peso (propiedad `layout_weight`) de 30 y al segundo de 70 para que la columna de listado ocupe un 30% de la pantalla a la izquierda y la de detalle ocupe el resto.

Por último, para el caso de pantalla grande vertical será prácticamente igual, sólo que usaremos un `LinearLayout` vertical.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

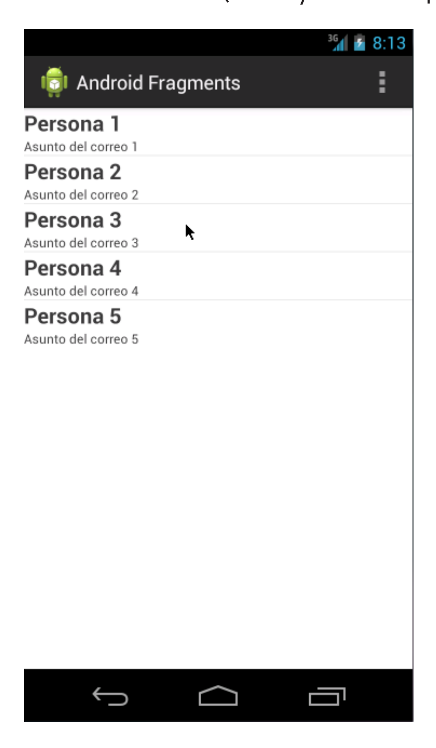
    <fragment class="net.sgoliver.android.fragments.FragmentListado"
        android:id="@+id/FrgListado"
        android:layout_weight="40"
        android:layout_width="match_parent"
        android:layout_height="0px" />

    <fragment class="net.sgoliver.android.fragments.FragmentDetalle"
        android:id="@+id/FrgDetalle"
        android:layout_weight="60"
        android:layout_width="match_parent"
        android:layout_height="0px" />

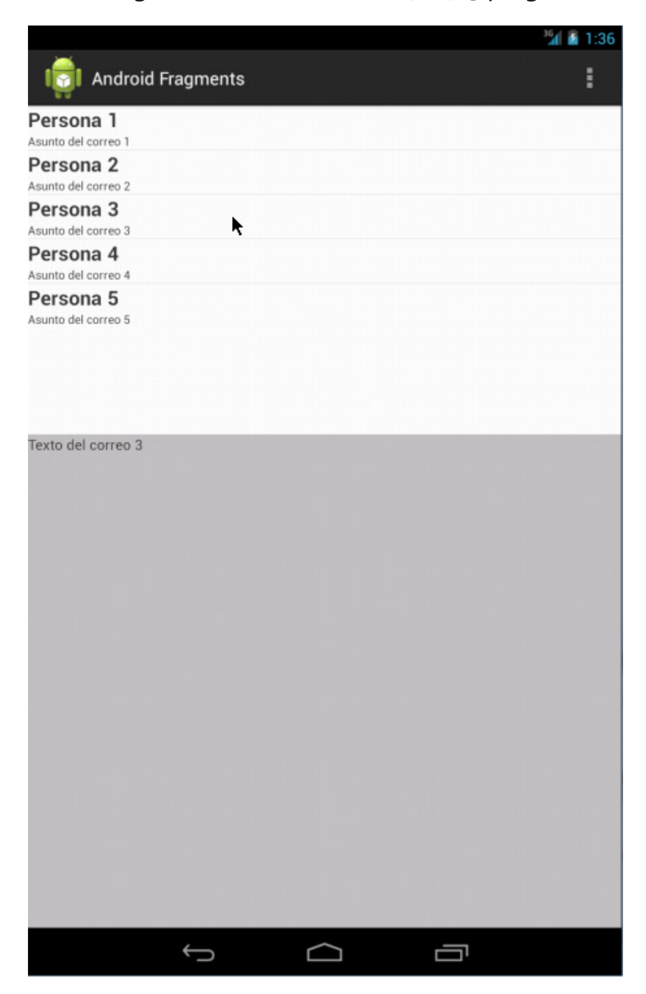
</LinearLayout>
```

Hecho esto, ya podríamos ejecutar la aplicación en el emulador y comprobar si se selecciona automáticamente el layout correcto dependiendo de las características del AVD que estemos utilizando. En mi caso he definido 2 AVD, uno con pantalla normal y otro grande al que durante las pruebas he modificado su orientación (pulsando Ctrl+F12). El resultado fue el siguiente:

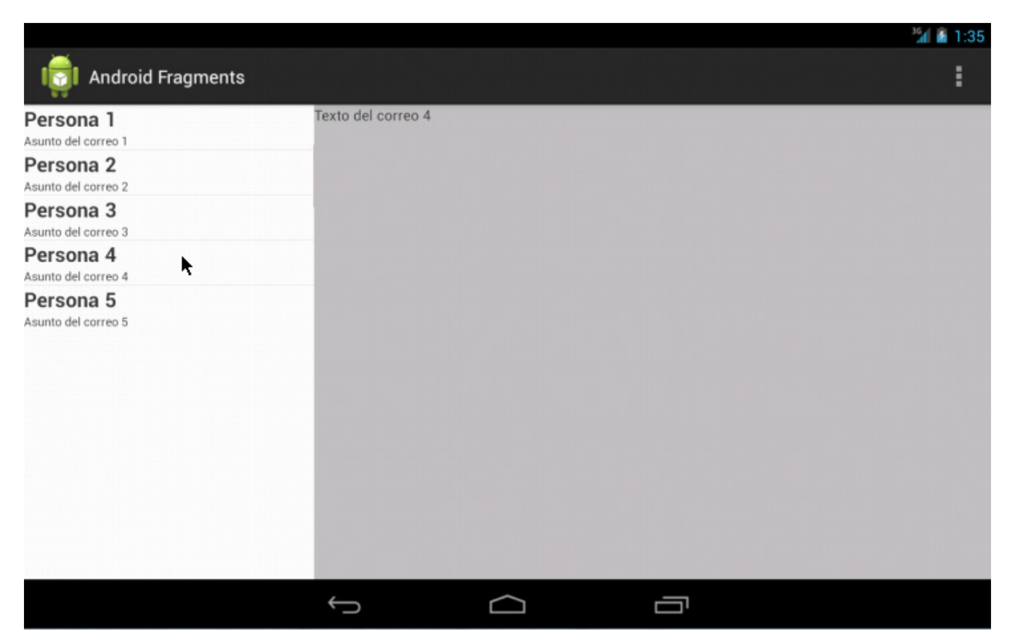
Pantalla normal (Galaxy Nexus – 4.7 pulgadas):



Pantalla grande vertical (Nexus 7 – 7.3 pulgadas):



Pantalla grande horizontal (Nexus 7 – 7.3 pulgadas):



Como vemos en las imágenes anteriores, la interfaz se ha adaptado perfectamente a la pantalla en cada caso, mostrándose uno o ambos fragments, y en caso de mostrarse ambos distribuyéndose horizontal o verticalmente.

Lo que aún no hemos implementado en la lógica de la aplicación es lo que debe ocurrir al pulsarse un elemento

de la lista de correos. Para ello, empezaremos asignando el evento `onItemClick()` a la lista dentro del método `onActivityCreated()` de la clase `FragmentListado`. Lo que hagamos al capturar este evento dependerá de si en la pantalla se está viendo el fragment de detalle o no:

Si existe el fragment de detalle habría que obtener una referencia a él y llamar a su método `mostrarDetalle()` con el texto del correo seleccionado.

En caso contrario, tendríamos que navegar a la actividad secundaria `DetalleActivity` para mostrar el detalle.

Sin embargo existe un problema, un fragment no tiene por qué conocer la existencia de ningún otro, es más, deberían diseñarse de tal forma que fueran lo más independientes posible, de forma que puedan reutilizarse en distintas situaciones sin problemas de dependencias con otros elementos de la interfaz. Por este motivo, el patrón utilizado normalmente en estas circunstancias no será tratar el evento en el propio fragment, sino definir y lanzar un evento personalizado al pulsarse el item de la lista y delegar a la actividad contenedora la lógica del evento, ya que ella sí debe conocer qué fragments componen su interfaz. ¿Cómo hacemos esto? Pues de forma análoga a cuando definimos eventos personalizados para un control. Definimos una interfaz con el método asociado al evento, en este caso llamada `CorreosListener` con un único método llamado `onCorreoSeleccionado()`, declaramos un atributo de la clase con esta interfaz y definimos un método `set…()` para poder asignar el evento desde fuera de la clase. Veamos cómo quedaría:

```
public class FragmentListado extends Fragment {

    //...

    private CorreosListener listener;

    //..

    @Override
    public void onActivityCreated(Bundle state) {
        super.onActivityCreated(state);

        lstListado = (ListView)getView().findViewById(R.id.LstListado);

        lstListado.setAdapter(new AdaptadorCorreos(this));

        lstListado.setOnItemClickListener(new OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> list, View view, int pos,
long id) {
                if (listener!=null) {
                    listener.onCorreoSeleccionado(
                        (Correo)lstListado.getAdapter().getItem(pos));
                }
            }
        });
    }

    public interface CorreosListener {
        void onCorreoSeleccionado(Correo c);
    }

    public void setCorreosListener(CorreosListener listener) {
        this.listener=listener;
    }
}
```

Como vemos, una vez definida toda esta parafernalia, lo único que deberemos hacer en el evento onItemClick() de la lista será lanzar nuestro evento personalizado onCorreoSeleccionado() pasándole como parámetro el contenido del correo, que en este caso obtendremos accediendo al adaptador con getAdapter() y recuperando el elemento con getItem().

Hecho esto, el siguiente paso será tratar este evento en la clase java de nuestra actividad principal. Para ello, en el onCreate() de nuestra actividad, obtendremos una referencia al fragment mediante el método getFragmentById() del fragment manager (componente encargado de gestionar los fragments) y asignaremos el evento mediante su método setCorreosListener() que acabamos de definir.

```
package net.sgoliver.android.fragments;

import net.sgoliver.android.fragments.FragmentListado.CorreosListener;
import android.content.Intent;
import android.os.Bundle;
import android.view.Menu;
import android.support.v4.app.FragmentActivity;

public class MainActivity extends FragmentActivity implements CorreosListener
{

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        FragmentListado frgListado
            =(FragmentListado)getSupportFragmentManager()
                .findFragmentById(R.id.FrgListado);

        frgListado.setCorreosListener(this);
    }

    @Override
    public void onCorreoSeleccionado(Correo c) {
        boolean hayDetalle =
            (getSupportFragmentManager()
                    .findFragmentById(R.id.FrgDetalle) != null);

        if(hayDetalle) {
            ((FragmentDetalle)getSupportFragmentManager()
                .findFragmentById(R.id.FrgDetalle))
                    .mostrarDetalle(c.getTexto());
        }
        else {
            Intent i = new Intent(this, DetalleActivity.class);
            i.putExtra(DetalleActivity.EXTRA_TEXTO, c.getTexto());
            startActivity(i);
        }
    }
}
```

Como vemos en el código anterior, en este caso hemos hecho que nuestra actividad herede de nuestra interfaz CorreosListener, por lo que nos basta pasar this al método setCorreosListener(). Adicionalmente, un detalle importante a destacar es que la actividad no hereda de la clase Activity como de costumbre, sino de FragmentActivity. Esto es así porque estamos utilizando la librería de compatibilidad android-support para utilizar fragments conservando la compatibilidad con versiones de Android anteriores a la 3.0. En caso de no necesitar esta compatibilidad podrías seguir heredando de

`Activity` sin problemas.

La mayor parte del interés de la clase anterior está en el método `onCorreoSeleccionado()`. Éste es el método que se ejecutará cuando el fragment de listado nos avise de que se ha seleccionado un determinado item de la lista. Esta vez sí, la lógica será la ya mencionada, es decir, si en la pantalla existe el fragment de detalle simplemente lo actualizaremos mediante `mostrarDetalle()` y en caso contrario navegaremos a la actividad `DetalleActivity`. Para este segundo caso, crearemos un nuevo `Intent` con la referencia a dicha clase, y le añadiremos como parámetro extra un campo de texto con el contenido del correo seleccionado. Finalmente llamamos a `startActivity()` para iniciar la nueva actividad.

Y ya sólo nos queda comentar la implementación de esta segunda actividad, `DetalleActivity`. El código será muy sencillo, y se limitará a recuperar el parámetro extra pasado desde la actividad anterior y mostrarlo en el fragment de detalle mediante su método `mostrarDetalle()`, todo ello dentro de su método `onCreate()`.

```
package net.sgoliver.android.fragments;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class DetalleActivity extends FragmentActivity {

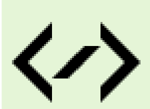
    public static final String EXTRA_TEXTO =
        "net.sgoliver.android.fragments.EXTRA_TEXTO";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_detalle);

        FragmentDetalle detalle =
            (FragmentDetalle)getSupportFragmentManager()
                .findFragmentById(R.id.FrgDetalle);

        detalle.mostrarDetalle(
            getIntent().getStringExtra(EXTRA_TEXTO));
    }
}
```

Y con esto finalizaríamos nuestro ejemplo. Si ahora volvemos a ejecutar la aplicación en el emulador podremos comprobar el funcionamiento de la selección en las distintas configuraciones de pantalla.

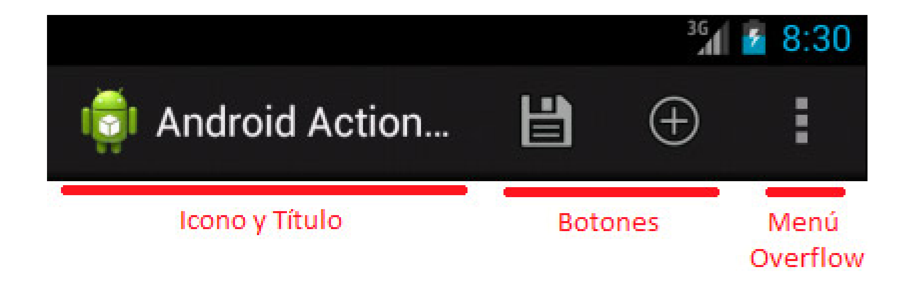


Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-fragments

Action Bar: Funcionamiento básico

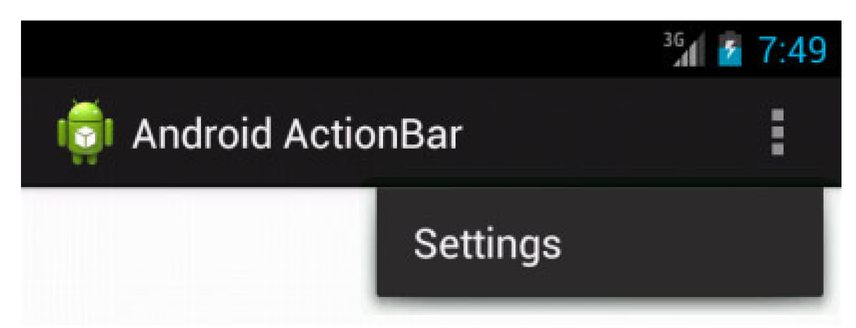
La *action bar* de Android es la barra de título y herramientas que aparece en la parte superior de muchas aplicaciones actuales. Normalmente muestra un icono, el título de la actividad en la que nos encontramos, una serie de botones de acción, y un menú desplegable (*menú de overflow*) donde se incluyen más acciones que no tienen espacio para mostrarse como botón o simplemente no se quieren mostrar como tal.



La action bar de Android es uno de esos componentes que Google no ha tratado demasiado bien al no incluirla en la librería de compatibilidad *android-support*. Esto significa que de forma nativa tan sólo es compatible con versiones de Android 3.0 o superiores. En este apartado nos centraremos únicamente en esta versión nativa de la plataforma (configuraremos nuestra aplicación de ejemplo para ejecutarse sobre Android 4).

Tan sólo a modo de referencia diré que existe una librería muy popular llamada *ActionBarSherlock* que proporciona una implementación alternativa de este componente que es compatible con versiónes de Android a partir de la 2.0. Otra alternativa para compatibilizar nuestras aplicaciones con versiones de Android anteriores a la 3.0 sería utilizar la implementación que Google proporciona como parte de los ejemplos de la plataforma, llamada `ActionBarCompat`, que puedes encontrar en la siguiente carpeta de tu instalación del SDK: `<carpeta-sdk>\samples\android-nn\ActionBarCompat`

Cuando se crea un proyecto con alguna de las últimas versiones del plugin ADT para Eclipse, la aplicación creada por defecto ya incluye "de serie" su action bar correspondiente. De hecho, si creamos un proyecto nuevo y directamente lo ejecutamos sobre un AVD con Android 3.0 o superior veremos como no solo se incluye la action bar sino que también aparece una acción "Settings" como muestra la siguiente imagen.



Vale, ¿pero donde está todo esto definido en nuestro proyecto? La action bar de Android toma su contenido de varios sitios diferentes. En primer lugar muestra el **icono de la aplicación** (definido en el AndroidManifest mediante el atributo `android:icon` del elemento `<application>`) y el **título de la actividad** actual (definido en el AndroidManifest mediante el atributo `android:label` de cada elemento `<activity>`). El resto de elementos se definen de la misma forma que los "antiguos" menús de aplicación que se utilizaban en versiones de Android 2.x e inferiores. De hecho, es exactamente la misma implementación. Nosotros definiremos un menú, y si la aplicación se ejecuta sobre Android 2.x las acciones se mostrarán como elementos del menú como tal (ya que la action bar no se verá al no ser compatible) y si se ejecuta sobre Android 3.0 o superior aparecerán como acciones de la action bar, ya sea en forma de botón de acción o incluidas en el menú de overflow. En definitiva, una bonita forma de mantener cierta compatibilidad con versiones anteriores de Android, aunque en unas ocasiones se muestre la action bar y en otras no.

Y bien, ¿cómo se define un menú de aplicación? Pues en el curso hay un capítulo dedicado exclusivamente a ello donde poder profundizar, pero de cualquier forma, en este apartado daré las directrices generales para definir uno sin mucha dificultad.

Un menú se define, como la mayoría de los recursos de Android, mediante un fichero XML, y se colocará en la carpeta `/res/menu`. El menú se definirá mediante un elemento raiz `<menu>` y contendrá una serie de elementos `<item>` que representarán cada una de las opciones. Los elementos `<item>` por su parte podrán incluir varios atributos que lo definan, entre los que destacan los siguientes:

- `android:id`. El ID identificativo del elemento, con el que podremos hacer referencia dicha opción.

- `android:title`. El texto que se visualizará para la opción.
- `android:icon`. El icono asociado a la acción.
- `android:showAsAction`. Si se está mostrando una action bar, este atributo indica si la opción de menú se mostrará como botón de acción o como parte del menú de overflow. Puede tomar varios valores:
    - `ifRoom`. Se mostrará como botón de acción sólo si hay espacio disponible.
    - `withText`. Se mostrará el texto de la opción junto al icono en el caso de que éste se esté mostrando como botón de acción.
    - `never`. La opción siempre se mostrará como parte del menú de overflow.
    - `always`. La opción siempre se mostrará como botón de acción. Este valor puede provocar que los elementos se solapen si no hay espacio suficiente para ellos.

Así, por ejemplo, si abrimos el menú definido por defecto en el proyecto nuevo (llamado normalmente `/res/menu/activity_main.xml` si no lo hemos cambiado de nombre durante la creación del proyecto) veremos el siguiente código:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/menu_settings"
        android:showAsAction="never"
        android:title="@string/menu_settings"/>

</menu>
```

Como vemos se define un menú con una única opción, con el texto "Settings" y con el atributo `showAsAction="never"` de forma que ésta siempre aparezca en el menú de overflow.

Esta opción por defecto se incluye solo a modo de ejemplo, por lo que podríamos eliminarla sin problemas para incluir las nuestras propias. En mi caso la voy a conservar pero voy a añadir dos más de ejemplo: "Save" y "New", la primera de ellas para que se muestre, si hay espacio, como botón con su icono correspondiente, y la segunda igual pero además acompañada de su título de acción:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/menu_settings"
        android:showAsAction="never"
        android:title="@string/menu_settings"/>

    <item
        android:id="@+id/menu_save"
        android:showAsAction="ifRoom"
        android:icon="@android:drawable/ic_menu_save"
        android:title="@string/menu_guardar"/>

    <item
        android:id="@+id/menu_new"
        android:showAsAction="ifRoom|withText"
        android:icon="@android:drawable/ic_menu_add"
        android:title="@string/menu_nuevo"/>
</menu>
```

Como podéis ver en la segunda opción, se pueden combinar varios valores de `showAsAction` utilizando el caracter "|".

Una vez definido el menú en su fichero XML correspondiente tan sólo queda asociarlo a nuestra actividad principal. Esto se realiza sobrescribiendo el método `OnCreateOptionsMenu()` de la actividad, dentro del cual lo único que tenemos que hacer normalmente es inflar el menú llamando al método `inflate()` pasándole como parámetros el ID del fichero XML donde se ha definido. Este trabajo suele venir hecho ya al crear un proyecto nuevo desde Eclipse:

```
public class MainActivity extends Activity {

    ...

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is
present.
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }
}
```

Ejecutemos la aplicación ahora a ver qué ocurre.



Como podemos observar, la opción "Settings" sigue estando dentro del menú de overflow, y ahora aparecen como botones de acción las dos opciones que hemos marcado como `showAsAction="ifRoom"`, pero para la segunda no aparece el texto. ¿Y por qué? Porque no hay espacio disponible con la pantalla en vertical. Pero si rotamos el emulador para ver qué ocurre con la pantalla en horizontal (pulsando Ctrl + F12) vemos lo siguiente:



Con la pantalla en horizontal sí se muestra el texto de la segunda opción, tal como habíamos solicitado con el valor `withText` del atributo `showAsAction`.

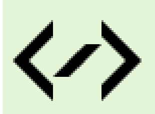
Ahora que ya sabemos definir los elementos de nuestra action bar queda saber cómo responder a las pulsaciones que haga el usuario sobre ellos. Para esto, al igual que se hace con los menús tradicionales (ver capítulo sobre menús para más detalles), sobrescribiremos el método `OnOptionsItemSelected()`, donde consultaremos la opción de menú que se ha pulsado mediante el método `getItemId()` de la opción de menú recibida como parámetro y actuaremos en consecuencia. En mi caso de ejemplo tan sólo escribiré un mensaje al log.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menu_new:
            Log.i("ActionBar", "Nuevo!");
            return true;
        case R.id.menu_save:
            Log.i("ActionBar", "Guardar!");;
            return true;
```

```
        case R.id.menu_settings:
            Log.i("ActionBar", "Settings!");;
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Si ejecutamos ahora la aplicación y miramos el log mientras pulsamos las distintas opciones de la action bar veremos como se muestran los mensajes definidos.

Y bien, esto es todo lo que habría que contar a nivel básico sobre la action bar, si no fuera porque los chicos de Android pensaron que también sería interesante "integrar" con ella de alguna forma la barra de pestañas, en caso de utilizarse este tipo de interfaz en la aplicación. Para no alargar demasiado este apartado, de momento pararemos aquí y dedicaré una segunda parte a este tema.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-actionbar

Action Bar: Tabs

Como comenté en el apartado anterior Android proporciona un mecanismo que nos permite "integrar" (por llamarlo de alguna manera) una barra de pestañas con la action bar.

En este momento muchos de vosotros seguro que habréis pensando: "*Yo ya he aprendido a usar el control `TabWidget` en un capítulo anterior de tu curso. ¿por qué iba a querer aprender otra forma de utilizar pestañas?*". Pues bien, además de convertirnos instantaneamente en programadores mucho más modernos y elegantes por utilizar fragments para nuestros tabs :), el hecho de enlazar una lista de pestañas con la action bar tiene una serie de ventajas adicionales, relacionadas sobre todo con la adaptación de tu interfaz a distintos tamaños y configuraciones de pantalla. Así, por ejemplo, si Android detecta que hay suficiente espacio disponible en la action bar, integrará las pestañas dentro de la propia action bar de forma que no ocupen espacio extra en la pantalla. Si por el contrario no hubiera espacio suficiente colocaría las pestañas bajo la action bar como de costumbre. Más tarde veremos un ejemplo gráfico de esto, pero ahora empecemos.

Lo primero que debemos hacer será crear un nuevo fragment para albergar el contenido de cada una de las pestañas que tendrá nuestra aplicación. Como ya vimos en el apartado dedicado a los fragments, necesitaremos crear como mínimo una pareja de ficheros para cada fragment, el primero para el layout XML y el segundo para su código java asociado. Para el caso de ejemplo de este apartado, que partirá del ya construido en el apartado anterior, incluiré tan sólo dos pestañas, y los ficheros asociados a ellas los llamaré de la siguiente forma:

Pestaña 1:

- Tab1Fragment.java
- fragment1.xml

Pestaña 2:

- Tab2Fragment.java
- fragment2.xml

La interfaz de los fragmets será mínima para no complicar el ejemplo, y contendrán únicamente una etiqueta

de texto "Tab 1" o "Tab 2" para poder diferenciarlas. Por ejemplo para la pestaña 1 tendríamos un fichero fragment1.xml con el siguiente contenido:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/tab_1" />

</LinearLayout>
```

Por su parte, su clase java asociada Tab1Fragment.java no tendrá ninguna funcionalidad, por lo que el código se limita a inflar el layout y poco más:

```
package net.sgoliver.android.actionbartabs;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class Tab1Fragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater,
            ViewGroup container, Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment1, container, false);
    }
}
```

Con esto ya tendríamos preparado el contenido de nuestras pestañas, y nos quedaría añadirlas a nuestra aplicación, enlazarlas con la action bar, y asignarles un listener desde el que poder responder a los eventos que se produzcan.

Vamos a empezar por el último paso, crear el listener. Como he comentado, este listener debe contener el código asociado a los eventos clásicos de las pestañas (normalmente la selección, reselección, o deselección de pestañas) y entre otras cosas tendrá que encargarse de mostrar en cada momento el fragment correspondiente a la pestaña seleccionada. Hay muchas formas de implementar este listener, y sé que la que yo voy a mostrar aquí no es la mejor de todas, pero sí es de las más sencillas para empezar. Vamos a crear nuestro listener creando una nueva clase que extienda de ActionBar.TabListener, en mi caso la llamaré MiTabListener (en un alarde de genialidad). Dentro de esta clase tendremos que sobrescribir los métodos de los eventos onTabSelected(), onTabUnselected() y onTabReselected(). Creo que el nombre de cada uno explica por sí solo lo que hacen. Veamos primero el código:

```java
package net.sgoliver.android.actionbartabs;

import android.app.ActionBar;
import android.app.Fragment;
import android.app.FragmentTransaction;
import android.app.ActionBar.Tab;
import android.util.Log;

public class MiTabListener implements ActionBar.TabListener {

    private Fragment fragment;

    public MiTabListener(Fragment fg)
    {
        this.fragment = fg;
    }

    @Override
    public void onTabReselected(Tab tab, FragmentTransaction ft) {
        Log.i("ActionBar", tab.getText() + " reseleccionada.");
    }

    @Override
    public void onTabSelected(Tab tab, FragmentTransaction ft) {
        Log.i("ActionBar", tab.getText() + " seleccionada.");
        ft.replace(R.id.contenedor, fragment);
    }

    @Override
    public void onTabUnselected(Tab tab, FragmentTransaction ft) {
        Log.i("ActionBar", tab.getText() + " deseleccionada.");
        ft.remove(fragment);
    }
}
```

Como podéis ver, en cada uno de estos métodos lo único que haremos en nuestro caso será mostrar u ocultar nuestros fragments de forma que quede visible el correspondiente a la pestaña seleccionada. Así, en el evento `onTabSelected()` reemplazaremos el fragment actualmente visible con el de la pestaña seleccionada (que será un atributo de nuestra clase, después veremos dónde y cómo se asigna), y en el método `onTabUnselected()` ocultamos el fragment asociado a la pestaña ya que está habrá sido deseleccionada. Ambas acciones se realizan llamando al método correspondiente de `FragmentTransaction`, que nos llega siempre como parámetro y nos permite gestionar los fragments de la actividad. En el primer caso se usará el método `replace()` y en el segundo el método `remove()`. Además de todo esto, he añadido mensajes de log en cada caso para poder comprobar que se lanzan los eventos de forma correcta.

Implementado el listener tan sólo nos queda crear las pestañas, asociarle sus fragments correspondientes, y enlazarlas a nuestra action bar. Todo esto lo haremos en el método `onCreate()` de la actividad principal. Son muchos pasos, pero sencillos todos ellos.

Comenzaremos obteniendo una referencia a la action bar mediante el método `getActionBar()`. Tras esto estableceremos su método de navegación a `NAVIGATION_MODE_TABS` para que sepa que debe mostrar las pestañas. A continuación creamos las pestañas el método `newTab()` de la action bar y estableceremos su texto con `setText()`. Lo siguiente será instanciar los dos fragments y asociarlos a cada pestaña a través de la clase listener que hemos creado, llamando para ello a `setTabListener()`. Y por último añadiremos las pestañas a la action bar mediante `addTab()`. Estoy seguro que con el código se entenderá mucho mejor:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    //Obtenemos una referencia a la actionbar
    ActionBar abar = getActionBar();

    //Establecemos el modo de navegación por pestañas
    abar.setNavigationMode(
        ActionBar.NAVIGATION_MODE_TABS);

    //Ocultamos si queremos el título de la actividad
    //abar.setDisplayShowTitleEnabled(false);

    //Creamos las pestañas
    ActionBar.Tab tab1 =
        abar.newTab().setText("Tab 1");

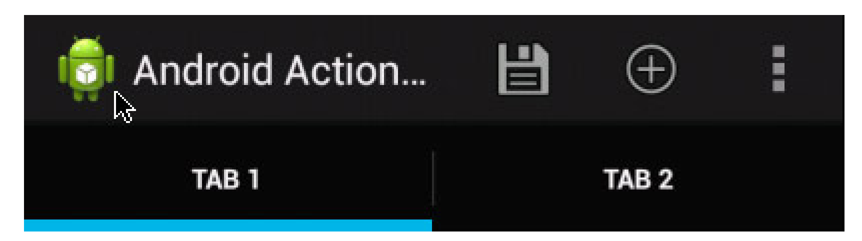
    ActionBar.Tab tab2 =
        abar.newTab().setText("Tab 2");

    //Creamos los fragments de cada pestaña
        Fragment tab1frag = new Tab1Fragment();
        Fragment tab2frag = new Tab2Fragment();

        //Asociamos los listener a las pestañas
        tab1.setTabListener(new MiTabListener(tab1frag));
        tab2.setTabListener(new MiTabListener(tab2frag));

        //Añadimos las pestañas a la action bar
        abar.addTab(tab1);
        abar.addTab(tab2);
}
```

Con esto habríamos acabado. Si ejecutamos la aplicación en el emulador veremos algo como lo siguiente:



Como vemos, Android ha colocado nuestras pestañas bajo la action bar porque no ha encontrado sitio suficiente arriba. Pero si probamos con la orientación horizontal (Ctrl+F12) pasa lo siguiente:

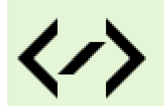


Dado que ya había espacio suficiente en la propia action bar, Android ha colocado las pestañas directamente sobre ella de forma que hemos ganado espacio para el resto de la interfaz. Esto no lo habríamos podido conseguir utilizando el control `TabWidget`.

Por último, si cambiamos varias veces de pestaña deberíamos comprobar que se muestra en cada caso el fragment correcto y que el en log aparecen los mensajes de depuración que hemos incluido.

Espero que con estos dos últimos apartados hayáis aprendido al menos a aprovechar la funcionalidad básica

de la action bar. Habría por supuesto más cosas que contar, por ejemplo algunas cuestiones de navegación entre actividades, pero eso lo dejaremos para más adelante.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-actionbar-tabs

3

Widgets

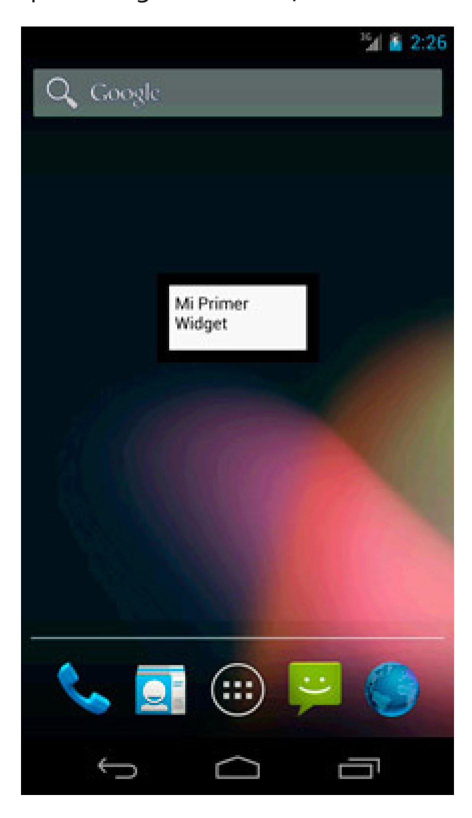
III. Widgets

Widgets básicos

En los dos próximos capítulos vamos a describir cómo crear un *widget* de escritorio (*home screen widget*).

En esta primera parte construiremos un *widget* estático (no será interactivo, ni contendrá datos actualizables, ni responderá a eventos) muy básico para entender claramente la estructura interna de un componente de este tipo, y en el siguiente capítulo completaremos el ejercicio añadiendo una ventana de configuración inicial para el widget, añadiremos algún dato que podamos actualizar periódicamente, y haremos que responda a pulsaciones del usuario.

Como hemos dicho, en esta primera parte vamos a crear un *widget* muy básico, consistente en un simple marco rectangular negro con un mensaje de texto predeterminado ("*Mi Primer Widget*"). La sencillez del ejemplo nos permitirá centrarnos en los pasos principales de la construcción de un widget Android y olvidarnos de otros detalles que nada tienen que ver con el tema que nos ocupa (gráficos, datos, ...). Para que os hagáis una idea, éste será el aspecto final de nuestro widget de ejemplo:



Los pasos principales para la creación de un widget Android son los siguientes:

1. Definición de su interfaz gráfica (*layout*).
2. Configuración XML del widget (`AppWidgetProviderInfo`).
3. Implementación de la funcionalidad del widget (`AppWidgetProvider`), especialmente su evento de actualización.
4. Declaración del widget en el *Android Manifest* de la aplicación.

En el primer paso no nos vamos a detener mucho ya que es análogo a cualquier definición de layout de las que hemos visto hasta ahora en el curso. En esta ocasión, la interfaz del widget estará compuesta únicamente por un par de *frames* (`FrameLayout`), uno negro exterior y uno blanco interior algo más pequeño para

simular el marco, y una etiqueta de texto (TextView) que albergará el mensaje a mostrar. Veamos cómo queda el layout xml, que para este ejemplo llamaremos "miwidget.xml":

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#000000"
    android:padding="10dp"
    android:layout_margin="5dp" >

    <FrameLayout android:id="@+id/frmWidget"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#FFFFFF"
        android:padding="5dp" >

        <TextView android:id="@+id/txtMensaje"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:textColor="#000000"
            android:text="@string/mi_primer_widget" />

    </FrameLayout>

</FrameLayout>
```

Cabe destacar aquí que, debido a que el layout de los widgets de Android está basado en un tipo especial de componentes llamados RemoteViews, no es posible utilizar en su interfaz todos los contenedores y controles que hemos visto en apartados anteriores sino sólo unos pocos básicos que se indican a continuación:

- Contenedores: FrameLayout, LinearLayout, RelativeLayout y GridLayout (éste ultimo a partir de Android 4).
- Controles: Button, ImageButton, ImageView, TextView, ProgressBar, Chronometer, AnalogClock y ViewFlipper. A partir de Android 3 también podemos utilizar ListView, GridView, StackView y AdapterViewFlipper, aunque su uso tiene algunas particularidades. En este apartado no trataremos este último caso, pero si necesitas información puedes empezar por la documentación oficial sobre el tema.

Aunque la lista de controles soportados no deja de ser curiosa (al menos en mi humilde opinión), debería ser suficiente para crear todo tipo de widgets.

Como segundo paso del proceso de construcción vamos a crear un nuevo fichero XML donde definiremos un conjunto de propiedades del widget, como por ejemplo su tamaño en pantalla o su frecuencia de actualización. Este XML se deberá crear en la carpeta \res\xml de nuestro proyecto.

En nuestro caso de ejemplo lo llamaremos "miwidget_wprovider.xml" y tendrá la siguiente estructura:

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:initialLayout="@layout/miwidget"
    android:minWidth="110dip"
    android:minHeight="40dip"
    android:label="@string/mi_primer_widget"
    android:updatePeriodMillis="3600000"
/>
```

Para nuestro widget estamos definiendo las siguientes propiedades:

- `initialLayout`: referencia al layout XML que hemos creado en el paso anterior.
- `minWidth`: ancho mínimo del widget en pantalla, en dp (*density-independent pixels*).
- `minHeight`: alto mínimo del widget en pantalla, en dp (*density-independent pixels*).
- `label`: nombre del widget que se mostrará en el menú de selección de Android.
- `updatePeriodMillis`: frecuencia de actualización del widget, en milisegundos.

Existen varias propiedades más que se pueden definir, por ejemplo el icono de vista previa del widget (`android:previewImage`, sólo para Android >3.0) o el indicativo de si el widget será redimensionable (`android:resizeMode`, sólo para Android >3.1) o la actividad de configuración del widget (`android:configure`). En el siguiente apartado utilizaremos alguna de ellas, el resto se pueden consultar en la documentación oficial de la clase `AppWidgetProviderInfo`.

Como sabemos, la pantalla inicial de Android se divide en un mínimo de 4×4 celdas (según el dispositivo pueden ser más) donde se pueden colocar aplicaciones, accesos directos y widgets. Teniendo en cuenta las diferentes dimensiones de estas celdas según el dispositivo y la orientación de la pantalla, existe una fórmula sencilla para ajustar las dimensiones de nuestro widget para que ocupe un número determinado de celdas sea cual sea la orientación:

- ancho_mínimo = (num_celdas * 70) – 30
- alto_mínimo = (num_celdas * 70) – 30

Atendiendo a esta fórmula, si queremos que nuestro widget ocupe por ejemplo un tamaño mínimo de 2 celdas de ancho por 1 celda de alto, deberemos indicar unas dimensiones de **110**dp x **40**dp.

Vamos ahora con el tercer paso. Éste consiste en implementar la funcionalidad de nuestro widget en su clase java asociada. Esta clase deberá heredar de `AppWidgetProvider`, que a su vez no es más que una clase auxiliar derivada de `BroadcastReceiver`, ya que los widgets de Android no son más que un caso particular de este tipo de componentes.

En esta clase deberemos implementar los mensajes a los que vamos a responder desde nuestro widget, entre los que destacan:

- `onEnabled()`: lanzado cuando se crea la primera instancia de un widget.
- `onUpdate()`: lanzado periódicamente cada vez que se debe actualizar un widget, por ejemplo cada vez que se cumple el periodo de tiempo definido por el parámetro `updatePeriodMillis` antes descrito, o cuando se añade el widget al escritorio.
- `onDeleted()`: lanzado cuando se elimina del escritorio una instancia de un widget.
- `onDisabled()`: lanzado cuando se elimina del escritorio la última instancia de un widget.

En la mayoría de los casos, tendremos que implementar como mínimo el evento `onUpdate()`. El resto de métodos dependerán de la funcionalidad de nuestro widget. En nuestro caso particular no nos hará falta ninguno de ellos ya que el widget que estamos creando no contiene ningún dato actualizable, por lo que crearemos la clase, llamada `MiWidget`, pero dejaremos vacío por el momento el método `onUpdate()`. En el siguiente apartado veremos qué cosas podemos hacer dentro de estos métodos.

```
package net.sgoliver.android.widgets;

import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.Context;

public class MiWidget extends AppWidgetProvider {
      @Override
    public void onUpdate(Context context,
                 AppWidgetManager appWidgetManager,
                 int[] appWidgetIds) {
        //Actualizar el widget
        //...
    }
}
```

El último paso del proceso será declarar el widget dentro del *manifest* de nuestra aplicación. Para ello, editaremos el fichero AndroidManifest.xml para incluir la siguiente declaración dentro del elemento `<application>`:

```
<application>
        ...
    <receiver android:name=".MiWidget" android:label="Mi Primer Widget">
        <intent-filter>
           <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
        </intent-filter>
        <meta-data
           android:name="android.appwidget.provider"
           android:resource="@xml/miwidget_wprovider" />
    </receiver>
</application>
```

El widget se declarará como un elemento `<receiver>` y deberemos aportar la siguiente información:

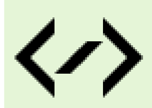
- Atributo `name`: Referencia a la clase java de nuestro widget, creada en el paso anterior.
- Elemento `<intent-filter>`, donde indicaremos los "eventos" a los que responderá nuestro widget, normalmente añadiremos el evento `APPWIDGET_UPDATE`, para detectar la acción de actualización.
- Elemento `<meta-data>`, donde haremos referencia con su atributo `resource` al XML de configuración que creamos en el segundo paso del proceso.

Con esto habríamos terminado de escribir los distintos elementos necesarios para hacer funcionar nuestro widget básico de ejemplo. Para probarlo, podemos ejecutar el proyecto de Eclipse en el emulador de Android, esperar a que se ejecute la aplicación principal (que estará vacía, ya que no hemos incluido ninguna funcionalidad para ella), ir a la pantalla principal del emulador y añadir nuestro widget al escritorio tal cómo lo haríamos en nuestro dispositivo físico.

- En **Android 2**: pulsación larga sobre el escritorio o tecla Menú, seleccionar la opción Widgets, y por último seleccionar nuestro Widget de la lista.
- En **Android 4**: accedemos al menú principal, pulsamos la pestaña *Widgets*, buscamos el nuestro en la lista y realizamos sobre él una pulsación larga hasta que el sistema nos deja arrastrarlo y colocarlo sobre el escritorio.

Con esto ya hemos conseguido la funcionalidad básica de un widget, es posible añadir varias instancias al escritorio, desplazarlos por la pantalla y eliminarlos enviándolos a la papelera.

En el próximo apartado veremos cómo podemos mejorar este widget añadiendo una pantalla de configuración inicial, mostraremos algún dato que se actualice periódicamente, y añadiremos la posibilidad de capturar eventos de pulsación sobre el widget.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-widgets-1

Widgets avanzados

Ya hemos visto cómo construir un widget básico para Android, y prometimos que dedicaríamos un apartado adicional a comentar algunas características más avanzadas de este tipo de componentes. Pues bien, en este segundo apartado sobre el tema vamos a ver cómo podemos añadir los siguientes elementos y funcionalidades al widget básico que ya construimos:

- Pantalla de configuración inicial.
- Datos actualizables de forma periódica.
- Eventos de usuario.

Como sabéis, intento simplificar al máximo todos los ejemplos que utilizo en este curso para que podamos centrar nuestra atención en los aspectos realmente importantes. En esta ocasión utilizaré el mismo criterio, y las únicas características (aunque suficientes para demostrar los tres conceptos anteriores) que añadiremos a nuestro widget serán las siguientes:

1. Añadiremos una pantalla de configuración inicial del widget, que aparecerá cada vez que se añada una nueva instancia del widget a nuestro escritorio. En esta pantalla podrá configurarse únicamente el mensaje de texto a mostrar en el widget.
2. Añadiremos un nuevo elemento de texto al widget que muestre la hora actual. Esto nos servirá para comprobar que el widget se actualiza periódicamente.
3. Añadiremos un botón al widget, que al ser pulsado forzará la actualización inmediata del mismo.

Empecemos por el primer punto, la pantalla de configuración inicial del widget. Y procederemos igual que para el diseño de cualquier otra actividad Android, definiendo su layout xml. En nuestro caso será muy sencilla, un cuadro de texto para introducir el mensaje a personalizar y dos botones, uno para aceptar la configuración y otro para cancelar (en cuyo caso el widget no se añade al escritorio). En esta ocasión llamaremos a este layout "`widget_config.xml`". Veamos cómo queda:

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

    <TextView android:id="@+id/LblMensaje"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="@string/mensaje_personalizado" />

    <EditText android:id="@+id/TxtMensaje"
              android:layout_height="wrap_content"
              android:layout_width="match_parent" />
```

```
    <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="horizontal" >

        <Button android:id="@+id/BtnAceptar"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/aceptar" />

        <Button android:id="@+id/BtnCancelar"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/cancelar" />

    </LinearLayout>
</LinearLayout>
```

Una vez diseñada la interfaz de nuestra actividad de configuración tendremos que implementar su funcionalidad en java. Llamaremos a la clase `WidgetConfig`, su estructura será análoga a la de cualquier actividad de Android, y las acciones a realizar serán las comentadas a continuación. En primer lugar nos hará falta el identificador de la instancia concreta del widget que se configurará con esta actividad. Este ID nos llega como parámetro del *intent* que ha lanzado la actividad. Como ya vimos en un apartado anterior del curso, este *intent* se puede recuperar mediante el método `getIntent()` y sus parámetros mediante el método `getExtras()`. Conseguida la lista de parámetros del intent, obtendremos el valor del ID del widget accediendo a la clave `AppWidgetManager.EXTRA_APPWIDGET_ID`. Veamos el código hasta este momento:

```
public class WidgetConfig extends Activity {
    private Button btnAceptar;
    private Button btnCancelar;
    private EditText txtMensaje;
    private int widgetId = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.widget_config);

        //Obtenemos el Intent que ha lanzado esta ventana
        //y recuperamos sus parámetros
        Intent intentOrigen = getIntent();
        Bundle params = intentOrigen.getExtras();

        //Obtenemos el ID del widget que se está configurando
        widgetId = params.getInt(
                AppWidgetManager.EXTRA_APPWIDGET_ID,
                AppWidgetManager.INVALID_APPWIDGET_ID);

        //Establecemos el resultado por defecto (si se pulsa el botón 'Atrás'
        //del teléfono será éste el resultado devuelto).
        setResult(RESULT_CANCELED);

        //...
    }
}
```

En el código también podemos ver como aprovechamos este momento para establecer el resultado por defecto a devolver por la actividad de configuración mediante el método `setResult()`. Esto es importante porque las actividades de configuración de widgets deben devolver siempre un resultado (`RESULT_OK` en caso de aceptarse la configuración, o `RESULT_CANCELED` en caso de salir de la configuración sin aceptar los cambios). Estableciendo aquí ya un resultado `RESULT_CANCELED` por defecto nos aseguramos de que si el usuario sale de la configuración pulsando el botón *Atrás* del teléfono no añadiremos el widget al escritorio, mismo resultado que si pulsáramos el botón "*Cancelar*" de nuestra actividad.

Como siguiente paso recuperamos las referencias a cada uno de los controles de la actividad de configuración:

```
//Obtenemos la referencia a los controles de la pantalla
btnAceptar = (Button)findViewById(R.id.BtnAceptar);
btnCancelar = (Button)findViewById(R.id.BtnCancelar);
txtMensaje = (EditText)findViewById(R.id.TxtMensaje);
```

Por último, implementaremos las acciones de los botones "*Aceptar*" y "*Cancelar*". En principio, el botón Cancelar no tendría por qué hacer nada, tan sólo finalizar la actividad mediante una llamada al método `finish()` ya que el resultado `CANCELED` ya se ha establecido por defecto anteriormente:

```
//Implementación del botón "Cancelar"
btnCancelar.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        //Devolvemos como resultado: CANCELAR (RESULT_CANCELED)
        finish();
    }
});
```

En el caso del botón Aceptar tendremos que hacer más cosas:

1. Guardar de alguna forma el mensaje que ha introducido el usuario.
2. Actualizar manualmente la interfaz del widget según la configuración establecida.
3. Devolver el resultado `RESULT_OK` aportando además el ID del widget.

Para el primer punto nos ayudaremos de la API de Preferencias (para más información leer el capítulo dedicado a este tema). En nuestro caso, guardaremos una sola preferencia cuya clave seguirá el patrón "`msg_`*`IdWidget`*", esto nos permitirá distinguir el mensaje configurado para cada instancia del widget que añadamos a nuestro escritorio de Android.

El segundo paso indicado es necesario debido a que si definimos una actividad de configuración para un widget, será ésta la que tenga la responsabilidad de realizar la primera actualización del mismo en caso de ser necesario. Es decir, tras salir de la actividad de configuración no se lanzará automáticamente el evento `onUpdate()` del widget (sí se lanzará posteriormente y de forma periódica según la configuración del parámetro `updatePeriodMillis` del provider que veremos más adelante), sino que tendrá que ser la propia actividad quien fuerce la primera actualización. Para ello, simplemente obtendremos una referencia al widget manager de nuestro contexto mediente el método `AppWidgetManager.getInstance()` y con esta referencia llamaremos al método estático de actualización del widget `MiWidget.actualizarWidget()`, que actualizará los datos de todos los controles del widget (lo veremos un poco más adelante).

Por último, al resultado a devolver (`RESULT_OK`) deberemos añadir información sobre el ID de nuestro widget. Esto lo conseguimos creando un nuevo Intent que contenga como parámetro el ID del widget que recuperamos antes y estableciéndolo como resultado de la actividad mediante el método `setResult(resultado, intent)`. Por último llamaremos al método `finish()` para finalizar la actividad. Con estas indicaciones, veamos cómo quedaría el código del botón *Aceptar*:

```
//Implementación del botón "Aceptar"
btnAceptar.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View arg0) {
            //Guardamos el mensaje personalizado en las preferencias
            SharedPreferences prefs =
                getSharedPreferences("WidgetPrefs", Context.MODE_PRIVATE);
            SharedPreferences.Editor editor = prefs.edit();

            editor.putString("msg_" + widgetId,
                             txtMensaje.getText().toString());
            editor.commit();

            //Actualizamos el widget tras la configuración
            AppWidgetManager appWidgetManager =
                AppWidgetManager.getInstance(WidgetConfig.this);
            MiWidget.actualizarWidget(WidgetConfig.this,
                                      appWidgetManager, widgetId);

            //Devolvemos como resultado: ACEPTAR (RESULT_OK)
            Intent resultado = new Intent();
            resultado.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, widgetId);
            setResult(RESULT_OK, resultado);
            finish();
        }
});
```

Ya hemos terminado de implementar nuestra actividad de configuración. Pero para su correcto funcionamiento aún nos quedan dos detalles más por modificar. En primer lugar tendremos que declarar esta actividad en nuestro fichero `AndroidManifest.xml`, indicando que debe responder a los mensajes de tipo `APPWIDGET_CONFIGURE`:

```
<activity android:name=".WidgetConfig">
    <intent-filter>
        <action android:name="android.apwidget.action.APPWIDGET_CONFIGURE"/>
    </intent-filter>
</activity>
```

Por último, debemos indicar en el XML de configuración de nuestro widget (`xml\miwidget_wprovider.xml`) que al añadir una instancia de este widget debe mostrarse la actividad de configuración que hemos creado. Esto se consigue estableciendo el atributo `android:configure` del provider. Aprovecharemos además este paso para establecer el tiempo de actualización automática del widget al mínimo permitido por este parámetro (30 minutos) y el tamaño del widget a 3x2 celdas. Veamos cómo quedaría finalmente:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:initialLayout="@layout/miwidget"
    android:minWidth="180dip"
    android:minHeight="110dip"
    android:label="@string/mi_primer_widget"
    android:updatePeriodMillis="3600000"
    android:configure="net.sgoliver.android.widgets.WidgetConfig"
/>
```

Con esto, ya tenemos todo listo para que al añadir nuestro widget al escritorio se muestre automáticamente la pantalla de configuración que hemos construido. Podemos ejecutar el proyecto en este punto y comprobar que todo funciona correctamente.

Como siguiente paso vamos a modificar el layout del widget que ya construimos en el apartado anterior para añadir una nueva etiqueta de texto donde mostraremos la hora actual, y un botón que nos servirá para forzar la actualización de los datos del widget:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#000000"
    android:padding="10dp"
    android:layout_margin="5dp" >

      <LinearLayout android:id="@+id/FrmWidget"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:background="#FFFFFF"
            android:padding="5dp"
            android:orientation="vertical">

            <TextView android:id="@+id/LblMensaje"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:textColor="#000000"
                android:text="" />

            <TextView android:id="@+id/LblHora"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:textColor="#000000"
                android:text="" />

            <Button android:id="@+id/BtnActualizar"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:textColor="#000000"
                android:text="@string/actualizar" />

        </LinearLayout>

</FrameLayout>
```

Hecho esto, tendremos que modificar la implementación de nuestro provider (`MiWidget.java`) para que en cada actualización del widget se actualicen sus controles con los datos correctos (recordemos que en el apartado anterior dejamos este evento de actualización vacío ya que no mostrábamos datos actualizables en el widget). Esto lo haremos dentro del evento `onUpdate()` de nuestro provider.

Como ya dijimos, los componentes de un widget se basan en un tipo especial de vistas que llamamos *Remote Views*. Pues bien, para acceder a la lista de estos componentes que constituyen la interfaz del widget construiremos un nuevo objeto `RemoteViews` a partir del ID del layout del widget. Obtenida la lista de componentes, tendremos disponibles una serie de métodos `set` (uno para cada tipo de datos básicos) para establecer las propiedades de cada control del widget. Estos métodos reciben como parámetros el ID del control, el nombre del método que queremos ejecutar sobre el control, y el valor a establecer. Además de estos métodos, contamos adicionalmente con una serie de métodos más específicos para establecer directamente el texto y otras propiedades sencillas de los controles `TextView`, `ImageView`, `ProgressBar` y `Chronometer`, como por ejemplo `setTextViewText(idControl, valor)` para establecer el texto de un control `TextView`. Pueden consultarse todos los métodos disponibles en la documentación oficial de la clase `RemoteViews`. De esta forma, si por ejemplo queremos establecer el texto del control cuyo id es `LblMensaje` haríamos lo siguiente:

```
RemoteViews controles = new RemoteViews(context.getPackageName(), R.layout.
miwidget);
controles.setTextViewText(R.id.LblMensaje, "Mensaje de prueba");
```

El proceso de actualización habrá que realizarlo por supuesto para todas las instancias del widget que se hayan añadido al escritorio. Recordemos aquí que el evento `onUpdate()` recibe como parámetro la lista de widgets que hay que actualizar.

Dicho esto, creo que ya podemos mostrar cómo quedaría el código de actualización de nuestro widget:

```
@Override
public void onUpdate(Context context,
                    AppWidgetManager appWidgetManager,
                    int[] appWidgetIds) {

    //Iteramos la lista de widgets en ejecución
    for (int i = 0; i < appWidgetIds.length; i++)
    {
        //ID del widget actual
        int widgetId = appWidgetIds[i];

        //Actualizamos el widget actual
        actualizarWidget(context, appWidgetManager, widgetId);
    }
}

public static void actualizarWidget(Context context,
                    AppWidgetManager appWidgetManager, int widgetId)
{
    //Recuperamos el mensaje personalizado para el widget actual
    SharedPreferences prefs =
            context.getSharedPreferences("WidgetPrefs", Context.MODE_PRIVATE);
    String mensaje = prefs.getString("msg_" + widgetId, "Hora actual:");

    //Obtenemos la lista de controles del widget actual
    RemoteViews controles =
            new RemoteViews(context.getPackageName(), R.layout.miwidget);

    //Actualizamos el mensaje en el control del widget
    controles.setTextViewText(R.id.LblMensaje, mensaje);

    //Obtenemos la hora actual
    Calendar calendario = new GregorianCalendar();
    String hora = calendario.getTime().toLocaleString();

    //Actualizamos la hora en el control del widget
    controles.setTextViewText(R.id.LblHora, hora);

    //Notificamos al manager de la actualización del widget actual
    appWidgetManager.updateAppWidget(widgetId, controles);
}
```

Como vemos, todo el trabajo de actualización para un widget lo hemos extraído a un método estático independiente, de forma que también podamos llamarlo desde otras partes de la aplicación (como hacemos por ejemplo desde la actividad de configuración para forzar la primera actualización del widget).

Además quiero destacar la última línea del código, donde llamamos al método `updateAppWidget()` del *widget manager*. Esto es importante y necesario, ya que de no hacerlo la actualización de los controles no se

reflejará correctamente en la interfaz del widget.

Tras esto, ya sólo nos queda implementar la funcionalidad del nuevo botón que hemos incluido en el widget para poder forzar la actualización del mismo. A los controles utilizados en los widgets de Android, que ya sabemos que son del tipo `RemoteView`, no podemos asociar eventos de la forma tradicional que hemos visto en múltiples ocasiones durante el curso. Sin embargo, en su lugar, tenemos la posibilidad de asociar a un evento (por ejemplo, el click sobre un botón) un determinado mensaje (*Pending Intent*) de tipo *broadcast* que será lanzado cada vez que se produzca dicho evento. Además, podremos configurar el widget (que como ya indicamos no es más que un componente de tipo *broadcast receiver*) para que capture esos mensajes, e implementar en su evento `onReceive()` las acciones necesarias a ejecutar tras capturar el mensaje. Con estas tres acciones simularemos la captura de eventos sobre controles de un widget.

Vamos por partes. En primer lugar hagamos que se lance un intent de tipo broadcast cada vez que se pulse el botón del widget. Para ello, en el método `actualizarWidget()` construiremos un nuevo Intent asociándole una acción personalizada, que en nuestro caso llamaremos por ejemplo "`net.sgoliver.android.widgets.ACTUALIZAR_WIDGET`" . Como parámetro del nuevo Intent insertaremos mediante `putExtra()` el ID del widget actual de forma que más tarde podamos saber el widget concreto que ha lanzado el mensaje (recordemos que podemos tener varias instancias del mismo widget en el escritorio). Por último crearemos el `PendingIntent` mediante el método `getBroadcast()` y lo asociaremos al evento `onClick` del control llamando a `setOnClickPendingIntent()` pasándole el ID del control, en nuestro caso el botón de "*Actualizar*". Veamos cómo queda todo esto dentro del método `actualizarWidget()`:

```
Intent intent = new Intent("net.sgoliver.android.widgets.ACTUALIZAR_WIDGET");
intent.putExtra(
           AppWidgetManager.EXTRA_APPWIDGET_ID, widgetId);

PendingIntent pendingIntent =
      PendingIntent.getBroadcast(context, widgetId,
            intent, PendingIntent.FLAG_UPDATE_CURRENT);

controles.setOnClickPendingIntent(R.id.BtnActualizar, pendingIntent);
```

También podemos hacer por ejemplo que si pulsamos en el resto del espacio del widget (el no ocupado por el botón) se abra automáticamente la actividad principal de nuestra aplicación. Se haría de forma análoga, con la única diferencia que en vez de utilizar `getBroadcast()` utilizaríamos `getActivity()` y el `Intent` lo construiríamos a partir de la clase de la actividad principal:

```
Intent intent2 = new Intent(context, MainActivity.class);
PendingIntent pendingIntent2 =
      PendingIntent.getActivity(context, widgetId,
            intent2, PendingIntent.FLAG_UPDATE_CURRENT);

controles.setOnClickPendingIntent(R.id.FrmWidget, pendingIntent2);
```

Ahora vamos a declarar en el *Android Manifest* este mensaje personalizado, de forma que el widget sea capaz de capturarlo. Para ello, añadiremos simplemente un nuevo elemento `<intent-filter>` con nuestro nombre de acción personalizado dentro del componente `<receiver>` que ya teníamos definido:

```
<receiver android:name=".MiWidget" android:label="Mi Primer Widget">
   <intent-filter>
      <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
   </intent-filter>
   <intent-filter>
      <action android:name="net.sgoliver.android.widgets.ACTUALIZAR_WIDGET"/>
   </intent-filter>
   <meta-data
         android:name="android.appwidget.provider"
         android:resource="@xml/miwidget_wprovider" />
</receiver>
```

Por último, vamos a implementar el evento `onReceive()` del widget para actuar en caso de recibir nuestro mensaje de actualización personalizado. Dentro de este evento comprobaremos si la acción del mensaje recibido es la nuestra, y en ese caso recuperaremos el ID del widget que lo ha lanzado, obtendremos una referencia al *widget manager*, y por último llamaremos a nuestro método estático de actualización pasándole estos datos.

```
@Override
public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals(
                "net.sgoliver.android.widgets.ACTUALIZAR_WIDGET")) {

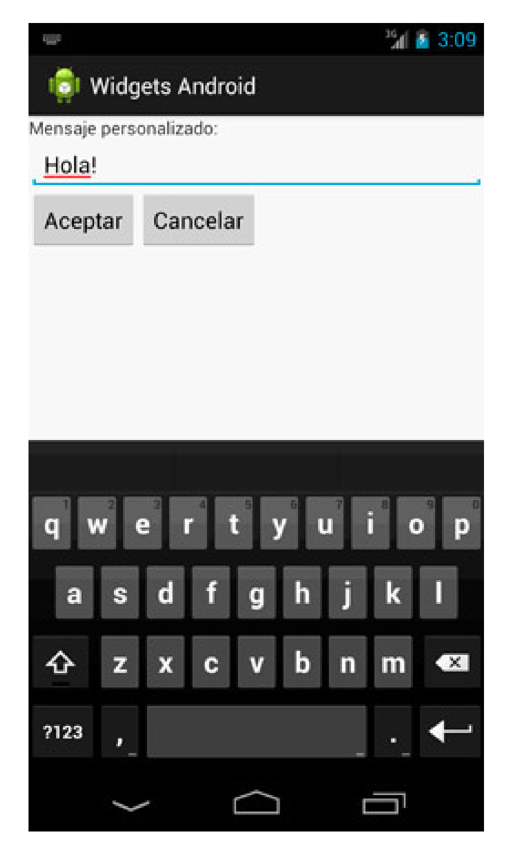
        //Obtenemos el ID del widget a actualizar
        int widgetId = intent.getIntExtra(
            AppWidgetManager.EXTRA_APPWIDGET_ID,
            AppWidgetManager.INVALID_APPWIDGET_ID);

        //Obtenemos el widget manager de nuestro contexto
        AppWidgetManager widgetManager =
            AppWidgetManager.getInstance(context);

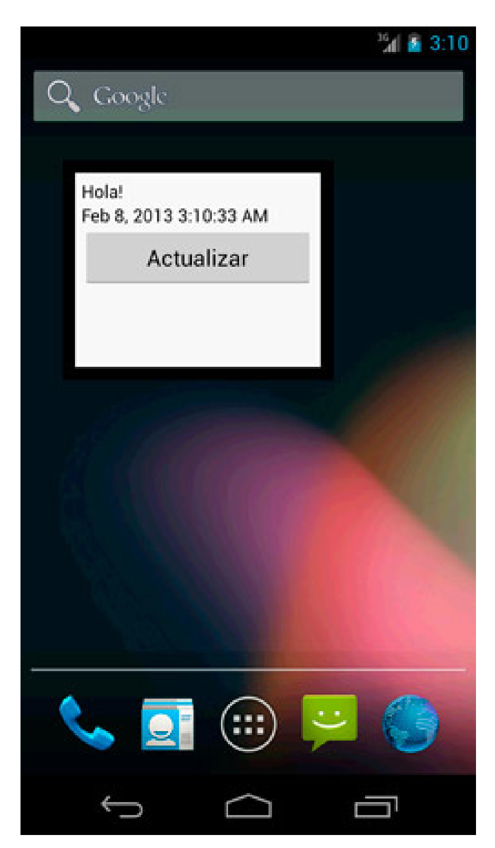
        //Actualizamos el widget
        if (widgetId != AppWidgetManager.INVALID_APPWIDGET_ID) {
            actualizarWidget(context, widgetManager, widgetId);
        }
}
```

Con esto, por fin, hemos ya finalizado la construcción de nuestro widget Android y podemos ejecutar el proyecto de Eclipse para comprobar que todo funciona correctamente, tanto para una sola instancia como para varias instancias simultaneas.

Cuando añadamos el widget al escritorio nos aparecerá la pantalla de configuración que hemos definido:

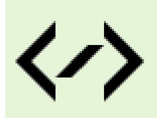


Una vez introducido el mensaje que queremos mostrar, pulsaremos el botón *Aceptar* y el widget aparecerá automáticamente en el escritorio con dicho mensaje, la fecha-hora actual y el botón *Actualizar*.



Un comentario final, la actualización automática del widget se ha establecido a la frecuencia mínima que permite el atributo `updatePeriodMillis` del widget provider, que son 30 minutos. Por tanto es difícil y aburrido esperar para verla en funcionamiento mientras probamos el widget en el emulador. Pero funciona, os lo aseguro. De cualquier forma, esos 30 minutos pueden ser un periodo demasiado largo de tiempo

según la funcionalidad que queramos dar a nuestro widget, que puede requerir tiempos de actualización mucho más cortos (ojo con el rendimiento y el gasto de batería). Para solucionar esto podemos hacer uso de Alarmas, pero por ahora no nos preocuparemos de esto.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-widgets-2

4

Menus

IV. Menús

Menús y Submenús básicos

En los dos siguientes apartados del curso nos vamos a centrar en la creación de menús de opciones en sus diferentes variantes.

NOTA IMPORTANTE: A partir de la versión 3 de Android los menús han caido en desuso debido a la aparición de la *Action Bar*. De hecho, si compilas tu aplicación con un target igual o superior a la API 11 (Android 3.0) verás como los menús que definas aparecen, no en su lugar habitual en la parte inferior de la pantalla, sino en el menú desplegable de la action bar en la parte superior derecha. Por todo esto, lo que leerás en este capítulo sobre menús y submenús aplica tan sólo a aplicaciones realizadas para Android 2.x o inferior. Si quieres conocer más detalles sobre la action bar tienes un capítulo dedicado a ello en el índice del curso. De cualquier forma, este capítulo sigue siendo útil ya que la forma de trabajar con la action bar se basa en la API de menús y en los recursos que comentaremos en este texto.

En Android podemos encontrar 3 tipos diferentes de menús:

- *Menús Principales*. Los más habituales, aparecen en la zona inferior de la pantalla al pulsar el botón 'menú' del teléfono.
- *Submenús*. Son menús secundarios que se pueden mostrar al pulsar sobre una opción de un menú principal.
- *Menús Contextuales*. Útiles en muchas ocasiones, aparecen al realizar una pulsación larga sobre algún elemento de la pantalla.

En este primer apartado sobre el tema veremos cómo trabajar con los dos primeros tipos de menús. En el siguiente, comentaremos los menús contextuales y algunas características más avanzadas.

Como casi siempre, vamos a tener dos alternativas a la hora de mostrar un menú en nuestra aplicación Android. La primera de ellas mediante la definición del menú en un fichero XML, y la segunda creando el menú directamente mediante código. En este apartado veremos ambas alternativas.

Veamos en primer lugar cómo crear un menú a partir de su definición en XML. Los ficheros XML de menú se deben colocar en la carpeta "`res\menu`" de nuestro proyecto y tendrán una estructura análoga a la del siguiente ejemplo (si hemos creado el proyecto con una versión reciente del plugin de Android para Eclipse es posible que ya tengamos un menú por defecto creado en esta carpeta, por lo que simplemente tendremos que modificarlo):

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <item android:id="@+id/MnuOpc1" android:title="Opcion1"
          android:icon="@android:drawable/ic_menu_preferences"></item>
    <item android:id="@+id/MnuOpc2" android:title="Opcion2"
          android:icon="@android:drawable/ic_menu_compass"></item>
    <item android:id="@+id/MnuOpc3" android:title="Opcion3"
          android:icon="@android:drawable/ic_menu_agenda"></item>
</menu>
```

Como vemos, la estructura básica de estos ficheros es muy sencilla. Tendremos un elemento principal `<menu>` que contendrá una serie de elementos `<item>` que se corresponderán con las distintas opciones a mostrar en el menú. Estos elementos `<item>` tendrán a su vez varias propiedades básicas, como su ID (`android:id`), su texto (`android:title`) o su icono (`android:icon`). Los iconos utilizados deberán estar por supuesto en las carpetas "`res\drawable-…`" de nuestro proyecto (al final del apartado os paso unos enlaces donde podéis conseguir algunos iconos de menú Android gratuitos) o bien utilizar alguno de los que ya vienen "de fábrica" con la plataforma Android. En nuestro caso de ejemplo he utilizado esta

segunda opción, por lo que haremos referencia a los recursos con el prefijo "@android:drawable/…". Si quisiéramos utilizar un recurso propio escribiríamos directamente "@drawable/…" seguido del nombre del recurso.

Una vez definido el menú en el fichero XML, tendremos que implementar el evento onCreateOptionsMenu() de la actividad que queremos que lo muestre. En este evento deberemos "inflar" el menú de forma parecida a como ya hemos hecho otras veces con otro tipo de layouts. Primero obtendremos una referencia al *inflater* mediante el método getMenuInflater() y posteriormente generaremos la estructura del menú llamando a su método infate() pasándole como parámetro el ID del menú definido en XML, que en nuestro caso será R.menu.menu_principal. Por último devolveremos el valor true para confirmar que debe mostrarse el menú.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    //Alternativa 1
    getMenuInflater().inflate(R.menu.activity_main, menu);
    return true;
}
```

Y ya hemos terminado, con estos sencillos pasos nuestra aplicación ya debería mostrar sin problemas el menú que hemos construido, aunque todavía nos faltaría implementar la funcionalidad de cada una de las opciones mostradas.



Como hemos comentado antes, este mismo menú también lo podríamos crear directamente mediante código, también desde el evento onCreateOptionsMenu(). Para ello, para añadir cada opción del menú podemos utilizar el método add() sobre el objeto de tipo Menu que nos llega como parámetro del evento. Este método recibe 4 parámetros: ID del grupo asociado a la opción (veremos qué es esto en el

siguiente apartado, por ahora utilizaremos Menu.NONE), un ID único para la opción (que declararemos como constantes de la clase), el orden de la opción (que no nos interesa por ahora, utilizaremos Menu.NONE) y el texto de la opción. Por otra parte, el icono de cada opción lo estableceremos mediante el método setIcon() pasándole el ID del recurso.

Veamos cómo quedaría el código utilizando esta alternativa, que generaría un menú exactamente igual al del ejemplo anterior:

```
private static final int MNU_OPC1 = 1;
private static final int MNU_OPC2 = 2;
private static final int MNU_OPC3 = 3;

//...

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    //Alternativa 2
    menu.add(Menu.NONE, MNU_OPC1, Menu.NONE, "Opcion1")
            .setIcon(android.R.drawable.ic_menu_preferences);
    menu.add(Menu.NONE, MNU_OPC2, Menu.NONE, "Opcion2")
            .setIcon(android.R.drawable.ic_menu_compass);
    menu.add(Menu.NONE, MNU_OPC3, Menu.NONE, "Opcion3")
            .setIcon(android.R.drawable.ic_menu_agenda);
    return true;
}
```

Construido el menú, la implementación de cada una de las opciones se incluirá en el evento onOptionsItemSelected() de la actividad que mostrará el menú. Este evento recibe como parámetro el item de menú que ha sido pulsado por el usuario, cuyo ID podemos recuperar con el método getItemId(). Según este ID podremos saber qué opción ha sido pulsada y ejecutar unas acciones u otras. En nuestro caso de ejemplo, lo único que haremos será modificar el texto de una etiqueta (lblMensaje) colocada en la pantalla principal de la aplicación.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.MnuOpc1:
            lblMensaje.setText("Opcion 1 pulsada!");
            return true;
        case R.id.MnuOpc2:
            lblMensaje.setText("Opcion 2 pulsada!");;
            return true;
        case R.id.MnuOpc3:
            lblMensaje.setText("Opcion 3 pulsada!");;
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

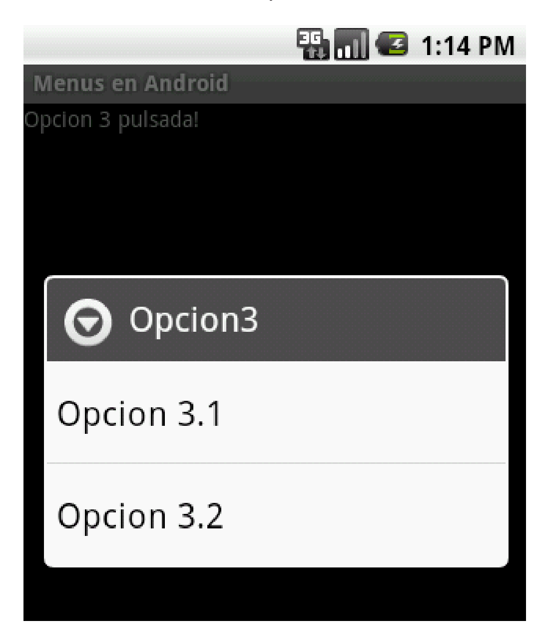
Ojo, el código anterior sería válido para el menú creado mediante XML. Si hubiéramos utilizado la implementación por código tendríamos que sustituir las constantes R.id.MnuOpc_ por nuestras constantes MNU_OPC_.

Con esto, hemos conseguido ya un menú completamente funcional. Si ejecutamos el proyecto en el emulador comprobaremos cómo al pulsar el botón de 'menú' del teléfono aparece el menú que hemos definido y que al pulsar cada opción se muestra el mensaje de ejemplo.

Pasemos ahora a comentar los submenús. Un submenú no es más que un menú secundario que se muestra al pulsar una opción determinada de un menú principal. Los submenús en Android se muestran en forma de lista emergente, cuyo título contiene el texto de la opción elegida en el menú principal. Como ejemplo, vamos a añadir un submenú a la *Opción 3* del ejemplo anterior, al que añadiremos dos nuevas opciones secundarias. Para ello, bastará con insertar en el XML de menú un nuevo elemento `<menu>` dentro del item correspondiente a la opción 3. De esta forma, el XML quedaría ahora como sigue:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <item android:id="@+id/MnuOpc1" android:title="Opcion1"
          android:icon="@android:drawable/ic_menu_preferences"></item>
    <item android:id="@+id/MnuOpc2" android:title="Opcion2"
          android:icon="@android:drawable/ic_menu_compass"></item>
    <item android:id="@+id/MnuOpc3" android:title="Opcion3"
          android:icon="@android:drawable/ic_menu_agenda">
        <menu>
            <item android:id="@+id/SubMnuOpc1"
                  android:title="Opcion 3.1" />
            <item android:id="@+id/SubMnuOpc2"
                  android:title="Opcion 3.2" />
        </menu>
    </item>
</menu>
```

Si volvemos a ejecutar ahora el proyecto y pulsamos la opción 3 nos aparecerá el correspondiente submenú con las dos nuevas opciones añadidas. Lo vemos en la siguiente imagen:



Comprobamos como efectivamente aparecen las dos nuevas opciones en la lista emergente, y que el título de la lista se corresponde con el texto de la opción elegida en el menú principal ("Opcion3").

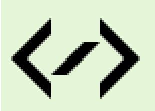
Para conseguir esto mismo mediante código procederíamos de forma similar a la anterior, con la única diferencia de que la opción de menú 3 la añadiremos utilizando el método `addSubMenu()` en vez de `add()`, y guardando una referencia al submenú. Sobre el submenú añadido insertaremos las dos nuevas opciones utilizando una vez más el método `add()`. Vemos cómo quedaría:

```
//Alternativa 2
menu.add(Menu.NONE, MNU_OPC1, Menu.NONE, "Opcion1")
    .setIcon(android.R.drawable.ic_menu_preferences);
menu.add(Menu.NONE, MNU_OPC2, Menu.NONE, "Opcion2")
    .setIcon(android.R.drawable.ic_menu_compass);

SubMenu smnu = menu.
    addSubMenu(Menu.NONE, MNU_OPC1, Menu.NONE, "Opcion3")
        .setIcon(android.R.drawable.ic_menu_agenda);
smnu.add(Menu.NONE, SMNU_OPC1, Menu.NONE, "Opcion 3.1");
smnu.add(Menu.NONE, SMNU_OPC2, Menu.NONE, "Opcion 3.2");
```

En cuanto a la implementación de estas opciones de submenú no habría diferencia con todo lo comentado anteriormente ya que también se tratan desde el evento `onOptionsItemSelected()`, identificándolas por su ID.

Por tanto, con esto habríamos terminado de comentar las opciones básicas a la hora de crear menús y submenús en nuestras aplicaciones Android. En el siguiente apartado veremos algunas opciones algo más avanzadas que, aunque menos frecuentes, puede que nos hagan falta para desarrollar determinadas aplicaciones.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-menus

Menús Contextuales

En el apartado anterior del curso ya vimos cómo crear menús y submenús básicos para nuestras aplicaciones Android. Sin embargo, existe otro tipo de menús que nos pueden ser muy útiles en determinados contextos: los *menús contextuales*. Este tipo de menú siempre va asociado a un control concreto de la pantalla y se muestra al realizar una pulsación larga sobre éste. Suele mostrar opciones específicas disponibles únicamente para el elemento pulsado. Por ejemplo, en un control de tipo lista podríamos tener un menú contextual que apareciera al pulsar sobre un elemento concreto de la lista y que permitiera editar su texto o eliminarlo de la colección.

NOTA [Android 3.0 o superior]: a diferencia de los menús de aplicación, para los que ya dijimos que entraron en desuso a partir de la versión 3 de Android en favor de la action bar, los menús contextuales pueden seguir utilizándose sin ningún problema aunque también se recomienda sustituirlos por acciones contextuales en la action bar, pero este tema no lo trataremos en este apartado.

Pues bien, la creación y utilización de este tipo de menús es muy parecida a lo que ya vimos para los menús y submenús básicos, pero presentan algunas particularidades que hacen interesante tratarlos al margen del resto en este nuevo apartado.

Empecemos por un caso sencillo. Vamos a partir de un proyecto nuevo, que ya debe contener por defecto una etiqueta de texto con un Hello World).

Vamos a añadir en primer lugar un menú contextual que aparezca al pulsar sobre la etiqueta de texto. Para ello, lo primero que vamos a hacer es indicar en el método `onCreate()` de nuestra actividad principal que la etiqueta tendrá asociado un menú contextual. Esto lo conseguimos con una llamada a `registerForContextMenu()`:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    //Obtenemos las referencias a los controles
    lblMensaje = (TextView)findViewById(R.id.LblMensaje);

    //Asociamos los menús contextuales a los controles
    registerForContextMenu(lblMensaje);
}
```

A continuación, igual que hacíamos con `onCreateOptionsMenu()` para los menús básicos, vamos a sobrescribir en nuestra actividad el evento encargado de construir los menús contextuales asociados a los diferentes controles de la aplicación. En este caso el evento se llama `onCreateContextMenu()`, y a diferencia de `onCreateOptionsMenu()` éste se llama cada vez que se necesita mostrar un menú contextual, y no una sola vez al inicio de la aplicación. En este evento actuaremos igual que para los menús básicos, inflando el menú XML que hayamos creado con las distintas opciones, o creando a mano el menú mediante el método `add()` [para más información leer el apartado anterior].

En nuestro ejemplo hemos definido un menú en XML llamado "`menu_ctx_etiqueta.xml`":

```
<?xml version="1.0" encoding="utf-8"?>
<menu
  xmlns:android="http://schemas.android.com/apk/res/android">

<item android:id="@+id/CtxLblOpc1"
    android:title="OpcEtiqueta1"></item>
<item android:id="@+id/CtxLblOpc2"
    android:title="OpcEtiqueta2"></item>

</menu>
```

Por su parte el evento `onCreateContextMenu()` quedaría de la siguiente forma:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                                ContextMenuInfo menuInfo)
{
    super.onCreateContextMenu(menu, v, menuInfo);

    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_ctx_etiqueta, menu);
}
```

Por último, para implementar las acciones a realizar tras pulsar una opción determinada del menú contextual vamos a implementar el evento `onContextItemSelected()` de forma análoga a cómo hacíamos con `onOptionsItemSelected()` para los menús básicos:

```
@Override
public boolean onContextItemSelected(MenuItem item) {

    switch (item.getItemId()) {
        case R.id.CtxLblOpc1:
            lblMensaje.setText("Etiqueta: Opcion 1 pulsada!");
            return true;
        case R.id.CtxLblOpc2:
            lblMensaje.setText("Etiqueta: Opcion 2 pulsada!");
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

Con esto, ya tendríamos listo nuestro menú contextual para la etiqueta de texto de la actividad principal, y como veis todo es prácticamente análogo a cómo construimos los menús y submenús básicos en el apartado anterior. En este punto ya podríamos ejecutar el proyecto en el emulador y comprobar su funcionamiento. Para ello, una vez iniciada la aplicación tan sólo tendremos que realizar una pulsación larga sobre la etiqueta de texto. En ese momento debería aparecer el menú contextual, donde podremos seleccionar cualquier de las dos opciones definidas.

Ahora vamos con algunas particularidades. Los menús contextuales se utilizan a menudo con controles de tipo lista, lo que añade algunos detalles que conviene mencionar. Para ello vamos a añadir a nuestro ejemplo una lista con varios datos de muestra y vamos a asociarle un nuevo menú contextual. Modificaremos el layout XML de la ventana principal para añadir el control `ListView` y modificaremos el método `onCreate()` para obtener la referencia al control, insertar varios datos de ejemplo, y asociarle un menú contextual:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    //Obtenemos las referencias a los controles
    lblMensaje = (TextView)findViewById(R.id.LblMensaje);
    lstLista = (ListView)findViewById(R.id.LstLista);

    //Rellenamos la lista con datos de ejemplo
    String[] datos =
        new String[]{"Elem1","Elem2","Elem3","Elem4","Elem5"};

    ArrayAdapter<String> adaptador =
        new ArrayAdapter<String>(this,
            android.R.layout.simple_list_item_1, datos);

    lstLista.setAdapter(adaptador);

    //Asociamos los menús contextuales a los controles
    registerForContextMenu(lblMensaje);
    registerForContextMenu(lstLista);
}
```

Como en el caso anterior, vamos a definir en XML otro menú para asociarlo a los elementos de la lista, lo llamaremos "`menu_ctx_lista`":

```
<?xml version="1.0" encoding="utf-8"?>
<menu
  xmlns:android="http://schemas.android.com/apk/res/android">

<item android:id="@+id/CtxLstOpc1"
    android:title="OpcLista1"></item>
<item android:id="@+id/CtxLstOpc2"
    android:title="OpcLista2"></item>

</menu>
```

Como siguiente paso, y dado que vamos a tener varios menús contextuales en la misma actividad, necesitaremos modificar el evento `onCreateContextMenu()` para que se construya un menú distinto dependiendo del control asociado. Esto lo haremos obteniendo el ID del control al que se va a asociar el menú contextual, que se recibe en forma de parámetro (`View v`) en el evento `onCreateContextMenu()`.

Utilizaremos para ello una llamada al método `getId()` de dicho parámetro:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                                ContextMenuInfo menuInfo)
{
    super.onCreateContextMenu(menu, v, menuInfo);

    MenuInflater inflater = getMenuInflater();

    if(v.getId() == R.id.LblMensaje)
        inflater.inflate(R.menu.menu_ctx_etiqueta, menu);
    else if(v.getId() == R.id.LstLista)
    {
        AdapterView.AdapterContextMenuInfo info =
            (AdapterView.AdapterContextMenuInfo)menuInfo;

        menu.setHeaderTitle(
            lstLista.getAdapter().getItem(info.position).toString());

        inflater.inflate(R.menu.menu_ctx_lista, menu);
    }
}
```

Vemos cómo en el caso del menú para el control lista hemos ido además un poco más allá, y hemos personalizado el título del menú contextual [mediante `setHeaderTitle()`] para que muestre el texto del elemento seleccionado en la lista. Para hacer esto nos hace falta saber la posición en la lista del elemento seleccionado, algo que podemos conseguir haciendo uso del último parámetro recibido en el evento `onCreateContextMenu()`, llamado `menuInfo`. Este parámetro contiene información adicional del control que se ha pulsado para mostrar el menú contextual, y en el caso particular del control `ListView` contiene la posición del elemento concreto de la lista que se ha pulsado. Para obtenerlo, convertimos el parámetro `menuInfo` a un objeto de tipo `AdapterContextMenuInfo` y accedemos a su atributo position tal como vemos en el código anterior.

La respuesta a este nuevo menú se realizará desde el mismo evento que el anterior, todo dentro de `onContextItemSelected()`. Por tanto, incluyendo las opciones del nuevo menú contextual para la lista el código nos quedaría de la siguiente forma:

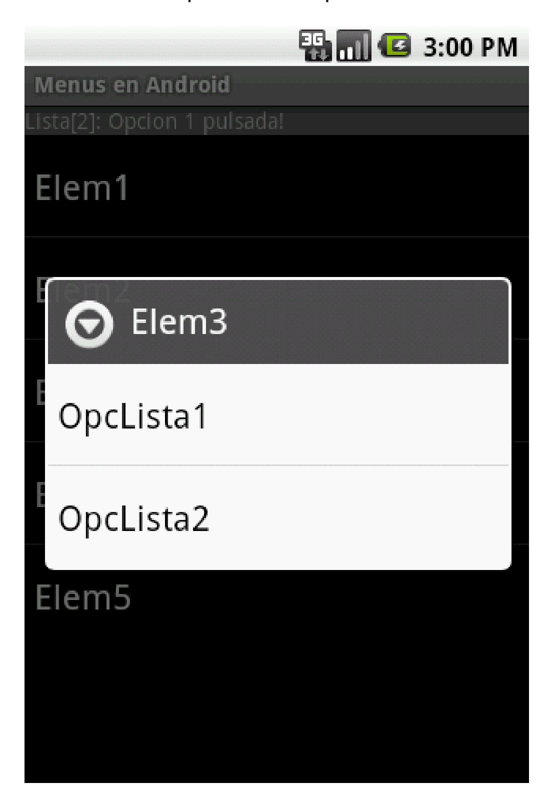
```
@Override
public boolean onContextItemSelected(MenuItem item) {

    AdapterContextMenuInfo info =
        (AdapterContextMenuInfo) item.getMenuInfo();

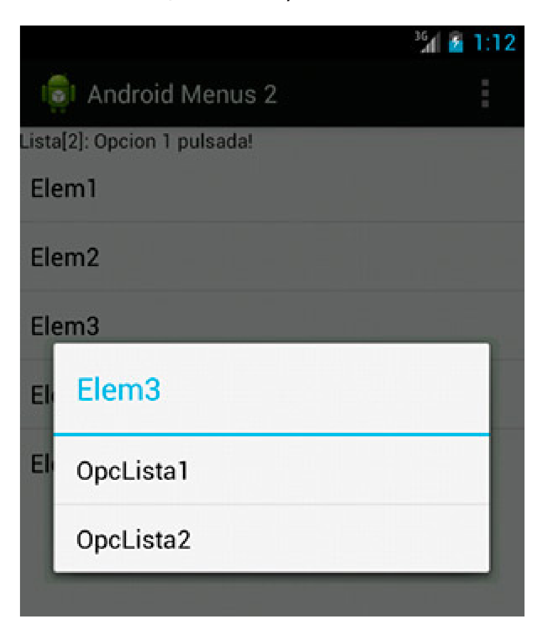
    switch (item.getItemId()) {
        case R.id.CtxLblOpc1:
            lblMensaje.setText("Etiqueta: Opcion 1 pulsada!");
            return true;
        case R.id.CtxLblOpc2:
            lblMensaje.setText("Etiqueta: Opcion 2 pulsada!");
            return true;
        case R.id.CtxLstOpc1:
            lblMensaje
                .setText("Lista[" + info.position + "]: Opcion 1 pulsada!");
            return true;
        case R.id.CtxLstOpc2:
            lblMensaje
                .setText("Lista[" + info.position + "]: Opcion 2 pulsada!");
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

Como vemos, aquí también utilizamos la información del objeto `AdapterContextMenuInfo` para saber qué elemento de la lista se ha pulsado, con la única diferencia de que en esta ocasión lo obtenemos mediante una llamada al método `getMenuInfo()` de la opción de menú (`MenuItem`) recibida como parámetro.

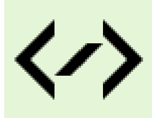
Si volvemos a ejecutar el proyecto en este punto podremos comprobar el aspecto de nuestro menú contextual al pulsar cualquier elemento de la lista:



En Android 4 sería muy similar:



A modo de resumen, en este apartado hemos visto cómo crear menús contextuales asociados a determinados elementos y controles de nuestra interfaz de la aplicación. Hemos visto cómo crear menús básicos y algunas particularidades que existen a la hora de asociar menús contextuales a elementos de un control de tipo lista. Para no alargar este apartado dedicaremos un tercero a comentar algunas opciones menos frecuentes, pero igualmente útiles, de los menús en Android.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-menus-2

Opciones avanzadas de menú

En los apartados anteriores del curso ya hemos visto cómo crear menús básicos para nuestras aplicaciones, tanto menús principales como de tipo contextual. Sin embargo, se nos quedaron en el tintero un par de temas que también nos pueden ser necesarios o interesantes a la hora de desarrollar una aplicación. Por un lado veremos los *grupos de opciones*, y por otro la *actualización dinámica* de un menú según determinadas condiciones.

Los grupos de opciones son un mecanismo que nos permite agrupar varios elementos de un menú de forma que podamos aplicarles ciertas acciones o asignarles determinadas características o funcionalidades de forma conjunta. De esta forma, podremos por ejemplo habilitar o deshabilitar al mismo tiempo un grupo de opciones de menú, o hacer que sólo se pueda seleccionar una de ellas. Lo veremos más adelante.

Veamos primero cómo definir un grupo de opciones de menú. Como ya comentamos, Android nos permite definir un menú de dos formas distintas: mediante un fichero XML, o directamente a través de código. Si elegimos la primera opción, para definir un grupo de opciones nos basta con colocar dicho grupo dentro de un elemento `<group>`, al que asignaremos un ID. Veamos un ejemplo. Vamos a definir un menú con 3 opciones principales, donde la última opción abre un submenú con 2 opciones que formen parte de un grupo. A todas las opciones le asignaremos un ID y un texto, y a las opciones principales asignaremos además una imagen.

```
<menu
  xmlns:android="http://schemas.android.com/apk/res/android">

<item android:id="@+id/MnuOpc1" android:title="Opcion1"
      android:icon="@android:drawable/ic_menu_preferences"></item>
<item android:id="@+id/MnuOpc2" android:title="Opcion2"
      android:icon="@android:drawable/ic_menu_compass"></item>
<item android:id="@+id/MnuOpc3" android:title="Opcion3"
      android:icon="@android:drawable/ic_menu_agenda">
    <menu>
        <group android:id="@+id/grupo1">

            <item android:id="@+id/SubMnuOpc1"
                  android:title="Opcion 3.1" />
            <item android:id="@+id/SubMnuOpc2"
                  android:title="Opcion 3.2" />

        </group>
    </menu>
</item>

</menu>
```

Como vemos, las dos opciones del submenú se han incluido dentro de un elemento <group>. Esto nos permitirá ejecutar algunas acciones sobre todas las opciones del grupo de forma conjunta, por ejemplo deshabilitarlas u ocultarlas:

```
//Deshabilitar todo el grupo
mnu.setGroupEnabled(R.id.grupo1, false);

//Ocultar todo el grupo
mnu.setGroupVisible(R.id.grupo1, false);
```

Además de estas acciones, también podemos modificar el comportamiento de las opciones del grupo de forma que tan sólo se pueda seleccionar una de ellas, o para que se puedan seleccionar varias. Con esto convertiríamos el grupo de opciones de menú en el equivalente a un conjunto de controles RadioButton o CheckBox respectivamente. Esto lo conseguimos utilizando el atributo android:checkableBehavior del elemento <group>, al que podemos asignar el valor "single" (selección exclusiva, tipo RadioButton) o "all" (selección múltiple, tipo CheckBox). En nuestro caso de ejemplo vamos a hacer seleccionable sólo una de las opciones del grupo:

```
<group android:id="@+id/grupo1" android:checkableBehavior="single">

    <item android:id="@+id/SubMnuOpc1"
        android:title="Opcion 3.1" />
    <item android:id="@+id/SubMnuOpc2"
        android:title="Opcion 3.2" />

</group>
```

Si optamos por construir el menú directamente mediante código debemos utilizar el método setGroupCheckable() al que pasaremos como parámetros el ID del grupo y el tipo de selección que deseamos (exclusiva o no). Así, veamos el método de construcción del menú anterior mediante código:

```
private static final int MNU_OPC1 = 1;
private static final int MNU_OPC2 = 2;
private static final int MNU_OPC3 = 3;
private static final int SMNU_OPC1 = 31;
private static final int SMNU_OPC2 = 32;

private static final int GRUPO_MENU_1 = 101;

private int opcionSeleccionada = 0;

//...

private void construirMenu(Menu menu)
{
     menu.add(Menu.NONE, MNU_OPC1, Menu.NONE, "Opcion1")
        .setIcon(android.R.drawable.ic_menu_preferences);
     menu.add(Menu.NONE, MNU_OPC2, Menu.NONE, "Opcion2")
        .setIcon(android.R.drawable.ic_menu_compass);

     SubMenu smnu = menu.addSubMenu(Menu.NONE, MNU_OPC3, Menu.NONE, "Opcion3")
                       .setIcon(android.R.drawable.ic_menu_agenda);

     smnu.add(GRUPO_MENU_1, SMNU_OPC1, Menu.NONE, "Opcion 3.1");
     smnu.add(GRUPO_MENU_1, SMNU_OPC2, Menu.NONE, "Opcion 3.2");

     //Establecemos la selección exclusiva para el grupo de opciones
     smnu.setGroupCheckable(GRUPO_MENU_1, true, true);

     //Marcamos la opción seleccionada actualmente
     if(opcionSeleccionada == 1)
          smnu.getItem(0).setChecked(true);
     else if(opcionSeleccionada == 2)
          smnu.getItem(1).setChecked(true);
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {

     construirMenu(menu);

     return true;
}
```

Como vemos, al final del método nos ocupamos de marcar manualmente la opción seleccionada actualmente, que debemos conservar en algún atributo interno (en mi caso lo he llamado `opcionSeleccionada`) de nuestra actividad. Esta marcación manual la hacemos mediante el método `getItem()` para obtener una opción determinada del submenú y sobre ésta el método `setChecked()` para establecer su estado. ¿Por qué debemos hacer esto? ¿No guarda Android el estado de las opciones de menú seleccionables? La respuesta es sí, sí lo hace, pero siempre que no reconstruyamos el menú entre una visualización y otra. ¿Pero no dijimos que la creación del menú sólo se realiza una vez en la primera llamada a `onCreateOptionsMenu()`? También es cierto, pero después veremos cómo también es posible preparar nuestra aplicación para poder modificar de forma dinámica un menú según determinadas condiciones, lo que sí podría implicar reconstruirlo previamente a cada visualización. En definitiva, si guardamos y restauramos nosotros mismos el estado de las opciones de menú seleccionables estaremos seguros de no perder su estado bajo ninguna circunstancia.

Por supuesto, para mantener el estado de las opciones hará falta actualizar el atributo `opcionSeleccionada` tras cada pulsación a una de las opciones. Esto lo haremos como siempre en el

método onOptionItemSelected().

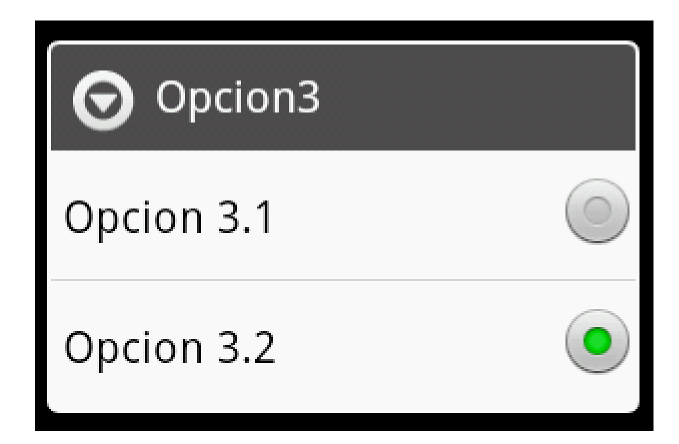
```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {

        //...
        //Omito el resto de opciones por simplicidad

        case SMNU_OPC1:
            opcionSeleccionada = 1;
            item.setChecked(true);
            return true;
        case SMNU_OPC2:
            opcionSeleccionada = 2;
            item.setChecked(true);
            return true;

        //...
    }
}
```

Con esto ya podríamos probar cómo nuestro menú funciona de la forma esperada, permitiendo marcar sólo una de las opciones del submenú. Si visualizamos y marcamos varias veces distintas opciones veremos cómo se mantiene correctamente el estado de cada una de ellas entre diferentes llamadas.



El segundo tema que quería desarrollar en este apartado trata sobre la modificación dinámica de un menú durante la ejecución de la aplicación de forma que éste sea distinto según determinadas condiciones. Supongamos por ejemplo que normalmente vamos a querer mostrar nuestro menú con 3 opciones, pero si tenemos marcada en pantalla una determinada opción queremos mostrar en el menú una opción adicional. ¿Cómo hacemos esto si dijimos que el evento onCreateOptionsMenu() se ejecuta una sola vez? Pues esto es posible ya que además del evento indicado existe otro llamado onPrepareOptionsMenu() que se ejecuta cada vez que se va a mostrar el menú de la aplicación, con lo que resulta el lugar ideal para adaptar nuestro menú a las condiciones actuales de la aplicación.

Para mostrar el funcionamiento de esto vamos a colocar en nuestra aplicación de ejemplo un nuevo *checkbox* (lo llamaré en mi caso chkMenuExtendido). Nuestra intención es que si este checkbox está marcado el menú muestre una cuarta opción adicional, y en caso contrario sólo muestre las tres opciones ya vistas en los ejemplos anteriores.

En primer lugar prepararemos el método construirMenu() para que reciba un parámetro adicional que indique si queremos construir un menú extendido o no, y sólo añadiremos la cuarta opción si este parámetro llega activado.

```
private void construirMenu(Menu menu, boolean extendido)
{
     menu.add(Menu.NONE, MNU_OPC1, Menu.NONE, "Opcion1")
        .setIcon(android.R.drawable.ic_menu_preferences);
     menu.add(Menu.NONE, MNU_OPC2, Menu.NONE, "Opcion2")
        .setIcon(android.R.drawable.ic_menu_compass);

     SubMenu smnu = menu.addSubMenu(Menu.NONE, MNU_OPC3, Menu.NONE, "Opcion3")
                       .setIcon(android.R.drawable.ic_menu_agenda);

     smnu.add(GRUPO_MENU_1, SMNU_OPC1, Menu.NONE, "Opcion 3.1");
     smnu.add(GRUPO_MENU_1, SMNU_OPC2, Menu.NONE, "Opcion 3.2");

     //Establecemos la selección exclusiva para el grupo de opciones
     smnu.setGroupCheckable(GRUPO_MENU_1, true, true);

     if(extendido)
        menu.add(Menu.NONE, MNU_OPC4, Menu.NONE, "Opcion4")
            .setIcon(android.R.drawable.ic_menu_camera);

     //Marcamos la opción seleccionada actualmente
     if(opcionSeleccionada == 1)
          smnu.getItem(0).setChecked(true);
     else if(opcionSeleccionada == 2)
          smnu.getItem(1).setChecked(true);
}
```

Además de esto, implementaremos el evento `onPrepareOptionsMenu()` para que llame a este método de una forma u otra dependiendo del estado del nuevo *checkbox*.

```
@Override
public boolean onPrepareOptionsMenu(Menu menu)
{
    menu.clear();

    if(chkMenuExtendido.isChecked())
        construirMenu(menu, true);
    else
        construirMenu(menu, false);

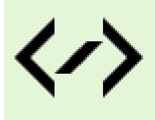
    return super.onPrepareOptionsMenu(menu);
}
```

Como vemos, en primer lugar debemos resetear el menú mediante el método `clear()` y posteriormente llamar de nuevo a nuestro método de construcción del menú indicando si queremos un menú extendido o no según el valor de la *check*.

Si ejecutamos nuevamente la aplicación de ejemplo, marcamos el checkbox y mostramos la tecla de menú podremos comprobar cómo se muestra correctamente la cuarta opción añadida.



Y con esto cerramos ya todos los temas referentes a menús que tenía intención de incluir en este Curso de Programación en Android. Espero que sea suficiente para cubrir las necesidades de muchas de vuestras aplicaciones.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-menus-3

5

Tratamiento de XML

V. Tratamiento de XML

Tratamiento de XML con SAX

En los siguientes apartados de este manual vamos a comentar las distintas posibilidades que tenemos a la hora de trabajar con datos en formato XML desde la plataforma Android.

A día de hoy, en casi todas las grandes plataformas de desarrollo existen varias formas de leer y escribir datos en formato XML. Los dos modelos más extendidos son SAX (*Simple API for XML*) y DOM (*Document Object Model*). Posteriormente, han ido apareciendo otros tantos, con más o menos éxito, entre los que destaca StAX (*Streaming API for XML*). Pues bien, Android no se queda atrás en este sentido e incluye estos tres modelos principales para el tratamiento de XML, o para ser más exactos, los dos primeros como tal y una versión análoga del tercero (*XmlPull*). Por supuesto con cualquiera de los tres modelos podemos hacer las mismas tareas, pero ya veremos cómo dependiendo de la naturaleza de la tarea que queramos realizar va a resultar más eficiente utilizar un modelo u otro.

Antes de empezar, unas anotaciones respecto a los ejemplos que voy a utilizar. Estas técnicas se pueden utilizar para tratar cualquier documento XML, tanto online como local, pero por utilizar algo conocido por la mayoría de vosotros todos los ejemplos van a trabajar sobre los datos XML de un documento RSS online, y en mi caso utilizaré como ejemplo el canal RSS de portada de europapress.com.

Un documento RSS de este *feed* tiene la estructura siguiente:

```
<rss version="2.0">
    <channel>
        <title>Europa Press</title>
        <link>http://www.europapress.es/</link>
        <description>Noticias de Portada.</description>
        <image>
            <url>http://s01.europapress.net/eplogo.gif</url>
            <title>Europa Press</title>
            <link>http://www.europapress.es</link>
        </image>
        <language>es-ES</language>
        <copyright>Copyright</copyright>
        <pubDate>Sat, 25 Dec 2010 23:27:26 GMT</pubDate>
        <lastBuildDate>Sat, 25 Dec 2010 22:47:14 GMT</lastBuildDate>
        <item>
            <title>Título de la noticia 1</title>
            <link>http://link_de_la_noticia_2.es</link>
            <description>Descripción de la noticia 2</description>
            <guid>http://identificador_de_la_noticia_2.es</guid>
            <pubDate>Fecha de publicación 2</pubDate>
        </item>
        <item>
            <title>Título de la noticia 2</title>
            <link>http://link_de_la_noticia_2.es</link>
            <description>Descripción de la noticia 2</description>
            <guid>http://identificador_de_la_noticia_2.es</guid>
            <pubDate>Fecha de publicación 2</pubDate>
        </item>
        ...
    </channel>
</rss>
```

Como puede observarse, se compone de un elemento principal `<channel>` seguido de varios datos relativos al canal y posteriormente una lista de elementos `<item>` para cada noticia con sus datos asociados.

En estos apartados vamos a describir cómo leer este XML mediante cada una de las tres alternativas citadas, y para ello lo primero que vamos a hacer es definir una clase java para almacenar los datos de una noticia. Nuestro objetivo final será devolver una lista de objetos de este tipo, con la información de todas las noticias. Por comodidad, vamos a almacenar todos los datos como cadenas de texto:

```
public class Noticia {

    private String titulo;
    private String link;
    private String descripcion;
    private String guid;
    private String fecha;

    public String getTitulo() {
        return titulo;
    }

    public String getLink() {
        return link;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public String getGuid() {
        return guid;
    }

    public String getFecha() {
        return fecha;
    }

    public void setTitulo(String t) {
        titulo = t;
    }

    public void setLink(String l) {
        link = l;
    }

    public void setDescripcion(String d) {
        descripcion = d;
    }

    public void setGuid(String g) {
        guid = g;
    }

    public void setFecha(String f) {
        fecha = f;
    }
}
```

Una vez conocemos la estructura del XML a leer y hemos definido las clases auxiliares que nos hacen falta para almacenar los datos, pasamos ya a comentar el primero de los modelos de tratamiento de XML.

SAX en Android

En el modelo SAX, el tratamiento de un XML se basa en un analizador (*parser*) que a medida que lee secuencialmente el documento XML va generando diferentes eventos con la información de cada elemento leído. Asi, por ejemplo, a medida que lee el XML, si encuentra el comienzo de una etiqueta `<title>` generará un evento de comienzo de etiqueta, `startElement()`, con su información asociada, si después de esa etiqueta encuentra un fragmento de texto generará un evento `characters()` con toda la información necesaria, y así sucesivamente hasta el final del documento. Nuestro trabajo consistirá por tanto en implementar las acciones necesarias a ejecutar para cada uno de los eventos posibles que se pueden generar durante la lectura del documento XML.

Los principales eventos que se pueden producir son los siguientes (consultar aquí la lista completa):

- `startDocument()`: comienza el documento XML.
- `endDocument()`: termina el documento XML.
- `startElement()`: comienza una etiqueta XML.
- `endElement()`: termina una etiqueta XML.
- `characters()`: fragmento de texto.

Todos estos métodos están definidos en la clase `org.xml.sax.helpers.DefaultHandler`, de la cual deberemos derivar una clase propia donde se sobrescriban los eventos necesarios. En nuestro caso vamos a llamarla `RssHandler`.

```
public class RssHandler extends DefaultHandler {
    private List<Noticia> noticias;
    private Noticia noticiaActual;
    private StringBuilder sbTexto;

    public List<Noticia> getNoticias(){
        return noticias;
    }

    @Override
    public void characters(char[] ch, int start, int length)
                   throws SAXException {

        super.characters(ch, start, length);

        if (this.notciaActual != null)
            builder.append(ch, start, length);
    }

    @Override
    public void endElement(String uri, String localName, String name)
                   throws SAXException {

        super.endElement(uri, localName, name);
```

```
        if (this.notciaActual != null) {

            if (localName.equals("title")) {
                noticiaActual.setTitulo(sbTexto.toString());
            } else if (localName.equals("link")) {
                noticiaActual.setLink(sbTexto.toString());
            } else if (localName.equals("description")) {
                noticiaActual.setDescripcion(sbTexto.toString());
            } else if (localName.equals("guid")) {
                noticiaActual.setGuid(sbTexto.toString());
            } else if (localName.equals("pubDate")) {
                noticiaActual.setFecha(sbTexto.toString());
            } else if (localName.equals("item")) {
                noticias.add(noticiaActual);
            }

            sbTexto.setLength(0);
        }
    }

    @Override
    public void startDocument() throws SAXException {

        super.startDocument();

        noticias = new ArrayList<Noticia>();
        sbTexto = new StringBuilder();
    }

    @Override
    public void startElement(String uri, String localName,
                   String name, Attributes attributes) throws SAXException {

        super.startElement(uri, localName, name, attributes);

        if (localName.equals("item")) {
            noticiaActual = new Noticia();
        }
    }
}
```

Como se puede observar en el código de anterior, lo primero que haremos será incluir como miembro de la clase la lista de noticias que pretendemos construir, `List<Noticia>` noticias, y un método `getNoticias()` que permita obtenerla tras la lectura completa del documento. Tras esto, implementamos directamente los eventos SAX necesarios.

Comencemos por `startDocument()`, este evento indica que se ha comenzado a leer el documento XML, por lo que lo aprovecharemos para inicializar la lista de noticias y las variables auxiliares.

Tras éste, el evento `startElement()` se lanza cada vez que se encuentra una nueva etiqueta de apertura. En nuestro caso, la única etiqueta que nos interesará será `<item>`, momento en el que inicializaremos un nuevo objeto auxiliar de tipo Noticia donde almacenaremos posteriormente los datos de la noticia actual.

El siguiente evento relevante es `characters()`, que se lanza cada vez que se encuentra un fragmento de texto en el interior de una etiqueta. La técnica aquí será ir acumulando en una variable auxiliar, `sbTexto`, todos los fragmentos de texto que encontremos hasta detectarse una etiqueta de cierre.

Por último, en el evento de cierre de etiqueta, `endElement()`, lo que haremos será almacenar en el atributo

apropiado del objeto `noticiaActual` (que conoceremos por el parámetro `localName` devuelto por el evento) el texto que hemos ido acumulando en la variable `sbTexto` y limpiaremos el contenido de dicha variable para comenzar a acumular el siguiente dato. El único caso especial será cuando detectemos el cierre de la etiqueta `<item>`, que significará que hemos terminado de leer todos los datos de la noticia y por tanto aprovecharemos para añadir la noticia actual a la lista de noticias que estamos construyendo.

Una vez implementado nuestro *handler*, vamos a crear una nueva clase que haga uso de él para parsear mediante SAX un documento XML concreto. A esta clase la llamaremos `RssParserSax`. Más adelante crearemos otras clases análogas a ésta que hagan lo mismo pero utilizando los otros dos métodos de tratamiento de XML ya mencionados. Esta clase tendrá únicamente un constructor que reciba como parámetro la URL del documento a parsear, y un método público llamado `parse()` para ejecutar la lectura del documento, y que devolverá como resultado una lista de noticias. Veamos cómo queda esta clase:

```
import java.io.IOException;
import java.io.InputStream;
import java.util.List;

import java.net.URL;
import javax.xml.parsers.SAXParser;
import java.net.MalformedURLException;
import javax.xml.parsers.SAXParserFactory;

public class RssParserSax
{
    private URL rssUrl;

    public RssParserSax(String url)
    {
        try
        {
            this.rssUrl = new URL(url);
        }
        catch (MalformedURLException e)
        {
            throw new RuntimeException(e);
        }
    }

    public List<Noticia> parse()
    {
        SAXParserFactory factory = SAXParserFactory.newInstance();

        try
        {
            SAXParser parser = factory.newSAXParser();
            RssHandler handler = new RssHandler();
            parser.parse(this.getInputStream(), handler);
            return handler.getNoticias();
        }
        catch (Exception e)
        {
            throw new RuntimeException(e);
        }
    }
```

```
    private InputStream getInputStream()
    {
        try
        {
            return rssUrl.openConnection().getInputStream();
        }
        catch (IOException e)
        {
            throw new RuntimeException(e);
        }
    }
}
```

Como se puede observar en el código anterior, el constructor de la clase se limitará a aceptar como parámetro la URL del documento XML a parsear a controlar la validez de dicha URL, generando una excepción en caso contrario.

Por su parte, el método `parse()` será el encargado de crear un nuevo parser SAX mediante sú fábrica correspondiente [lo que se consigue obteniendo una instancia de la fábrica con `SAXParserFactory.newInstance()` y creando un nuevo parser con `factory.newSaxParser()`] y de iniciar el proceso pasando al parser una instancia del *handler* que hemos creado anteriormente y una referencia al documento a parsear en forma de *stream*.

Para esto último, nos apoyamos en un método privado auxiliar `getInputStream()`, que se encarga de abrir la conexión con la URL especificada [mediante `openConnection()`] y obtener el *stream* de entrada [mediante `getInputStream()`].

Con esto ya tenemos nuestra aplicación Android preparada para parsear un documento XML online utilizando el modelo SAX. Veamos lo simple que sería ahora llamar a este parser por ejemplo desde nuestra actividad principal. Como ejemplo de tratamiento de los datos obtenidos mostraremos los titulares de las noticias en un cuadro de texto (`txtResultado`).

```
public void onCreate(Bundle savedInstanceState)
{
     super.onCreate(savedInstanceState);
     setContentView(R.layout.main);

     RssParserSax saxparser =
            new RssParserSax("http://www.europapress.es/rss/rss.aspx");

     List<Noticia> noticias = saxparser.parse();

     //Tratamos la lista de noticias
     //Por ejemplo: escribimos los títulos en pantalla
     txtResultado.setText("");
     for(int i=0; i<noticias.size(); i++)
     {
          txtResultado.setText(
                  txtResultado.getText().toString() +
            System.getProperty("line.separator") +
            noticias.get(i).getTitulo());
     }
}
```

Las líneas 6 y 9 del código anterior son las que hacen toda la magia. Primero creamos el parser SAX pasándole la URL del documento XML y posteriormente llamamos al método `parse()` para obtener una lista de objetos de tipo `Noticia` que posteriormente podremos manipular de la forma que queramos. Así

de sencillo.

Adicionalmente, para que este ejemplo funcione debemos añadir previamente permisos de acceso a internet para la aplicación. Esto se hace en el fichero AndroidManifest.xml, que quedaría de la siguiente forma:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="net.sgoliver"
      android:versionCode="1"
      android:versionName="1.0">

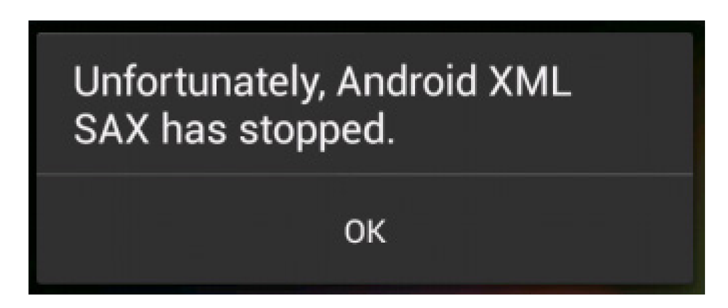
    <uses-permission
        android:name="android.permission.INTERNET" />

    <application android:icon="@drawable/icon"
                    android:label="@string/app_name">
        <activity android:name=".AndroidXml"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    </application>
    <uses-sdk android:minSdkVersion="7" />
</manifest>
```

En la línea 6 del código podéis ver cómo añadimos el permiso de acceso a la red mediante el elemento <uses-permission> con el parámetro android.permission.INTERNET

Pero hay algo más, si ejecutamos el código anterior en una versión de Android anterior a la 3.0 no tendremos ningún problema. La aplicación descargará el XML y lo parseará tal como hemos definido. Sin embargo, si utilizamos una versión de Android 3.0 o superior nos encontraremos con algo similar a esto:



Y si miramos el log de ejecución veremos que se ha generado una excepción del tipo NetworkOnMainThreadException. ¿Qué ha ocurrido? Pues bien, lo que ha ocurrido es que desde la versión 3 de Android se ha establecido una política que no permite a las aplicaciones ejecutar operaciones de larga duración en el hilo principal que puedan bloquear temporalmente la interfaz de usuario. En este caso, nosotros hemos intentado hacer una conexión a la red para descargarnos el XML y Android se ha quejado de ello generando un error.

¿Y cómo arreglamos esto? Pues la solución pasa por mover toda la lógica de descarga y lectura del XML a otro hilo de ejecución secundario, es decir, hacer el trabajo duro en segundo plano dejando libre el hilo principal de la aplicación y por tanto sin afectar a la interfaz. La forma más sencilla de hacer esto en Android es mediante la utilización de las llamadas AsyncTask o *tareas asíncronas*. Más adelante dedicaremos todo un capítulo a este tema, por lo que ahora no entraremos en detalle y nos limitaremos a ver cómo transformar nuestro código anterior para que funcione en versiones actuales de Android.

Lo que haremos será definir una nueva clase que extienda de AsyncTask y sobrescribiremos sus métodos

doInBackground() y onPostExecute().Al primero de ellos moveremos la descarga y parseo del XML, y al segundo las tareas que queremos realizar cuando haya finalizado el primero, en nuestro caso de ejemplo la escritura de los titulares al cuadro de texto de resultado. Quedaría de la siguiente forma:

```
//Tarea Asíncrona para cargar un XML en segundo plano
private class CargarXmlTask extends AsyncTask<String,Integer,Boolean> {

    protected Boolean doInBackground(String... params) {

      RssParserSax saxparser =
            new RssParserSax(params[0]);

      noticias = saxparser.parse();

       return true;
    }

    protected void onPostExecute(Boolean result) {

      //Tratamos la lista de noticias
      //Por ejemplo: escribimos los títulos en pantalla
      txtResultado.setText("");
       for(int i=0; i<noticias.size(); i++)
       {
            txtResultado.setText(
                  txtResultado.getText().toString() +
                        System.getProperty("line.separator") +
                        noticias.get(i).getTitulo());
        }
    }
}
```

Por último, sustituiremos el código de nuestra actividad principal por una simple llamada para crear y ejecutar la tarea en segundo plano:

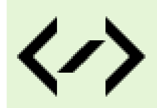
```
public void onCreate(Bundle savedInstanceState)
{
     super.onCreate(savedInstanceState);
     setContentView(R.layout.main);

     //Carga del XML mediante la tarea asíncrona

     CargarXmlTask tarea = new CargarXmlTask();
     tarea.execute("http://www.europapress.es/rss/rss.aspx");
}
```

Y ahora sí. Con esta ligera modificación del código nuestra aplicación se ejecutará correctamente en cualquier versión de Android.

En los siguientes apartados veremos los otros dos métodos de tratamiento XML en Android que hemos comentado (DOM y StAX) y por último intentaremos comentar las diferencias entre ellos dependiendo del contexto de la aplicación.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-xml-sax

Tratamiento de XML con SAX Simplificado

En el apartado anterior del tutorial vimos cómo realizar la lectura y tratamiento de un documento XML utilizando el modelo SAX clásico. Vimos cómo implementar un *handler* SAX, donde se definían las acciones a realizar tras recibirse cada uno de los posibles eventos generados por el parser XML.

Este modelo, a pesar de funcionar perfectamente y de forma bastante eficiente, tiene claras desventajas. Por un lado se hace necesario definir una clase independiente para el handler. Adicionalmente, la naturaleza del modelo SAX implica la necesidad de poner bastante atención a la hora de definir dicho handler, ya que los eventos SAX definidos no están ligados de ninguna forma a etiquetas concretas del documento XML sino que se lanzarán para todas ellas, algo que obliga entre otras cosas a realizar la distinción entre etiquetas dentro de cada evento y a realizar otros chequeos adicionales.

Estos problemas se pueden observar perfectamente en el evento `endElement()` que definimos en el ejemplo del apartado anterior. En primer lugar teníamos que comprobar la condición de que el atributo `noticiaActual` no fuera `null`, para evitar confundir el elemento `<title>` descendiente de `<channel>` con el del mismo nombre pero descendiente de `<item>`. Posteriormente, teníamos que distinguir con un *IF* gigantesco entre todas las etiquetas posibles para realizar una acción u otra. Y todo esto para un documento XML bastante sencillo. No es difícil darse cuenta de que para un documento XML algo más elaborado la complejidad del handler podría dispararse rápidamente, dando lugar a posibles errores.

Para evitar estos problemas, Android propone una variante del modelo SAX que evita definir una clase separada para el handler y que permite asociar directamente las acciones a etiquetas concretas dentro de la estructura del documento XML, lo que alivia en gran medida los inconvenientes mencionados.

Veamos cómo queda nuestro parser XML utilizando esta variante simplificada de SAX para Android y después comentaremos los aspectos más importantes del mismo.

```
import java.io.IOException;
import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import org.xml.sax.Attributes;
import android.sax.Element;
import android.sax.EndElementListener;
import android.sax.EndTextElementListener;
import android.sax.RootElement;
import android.sax.StartElementListener;
import android.util.Xml;

public class RssParserSax2
{
    private URL rssUrl;
    private Noticia noticiaActual;
```

```
    public RssParserSax2(String url)
    {
        try
        {
            this.rssUrl = new URL(url);
        } catch (MalformedURLException e)
        {
            throw new RuntimeException(e);
        }
    }

    public List<Noticia> parse()
    {
        final List<Noticia> noticias = new ArrayList<Noticia>();

        RootElement root = new RootElement("rss");
        Element channel = root.getChild("channel");
        Element item = channel.getChild("item");

        item.setStartElementListener(new StartElementListener(){
            public void start(Attributes attrs) {
                noticiaActual = new Noticia();
            }
        });

        item.setEndElementListener(new EndElementListener(){
            public void end() {
                noticias.add(noticiaActual);
            }
        });

        item.getChild("title").setEndTextElementListener(
            new EndTextElementListener(){
                public void end(String body) {
                    noticiaActual.setTitulo(body);
                }
        });

        item.getChild("link").setEndTextElementListener(
            new EndTextElementListener(){
                public void end(String body) {
                    noticiaActual.setLink(body);
                }
        });

        item.getChild("description").setEndTextElementListener(
            new EndTextElementListener(){
                public void end(String body) {
                    noticiaActual.setDescripcion(body);
                }
        });

        item.getChild("guid").setEndTextElementListener(
            new EndTextElementListener(){
                public void end(String body) {
                    noticiaActual.setGuid(body);
                }
        });
```

```
        item.getChild("pubDate").setEndTextElementListener(
            new EndTextElementListener(){
                public void end(String body) {
                    noticiaActual.setFecha(body);
                }
        });

        try
        {
            Xml.parse(this.getInputStream(),
                    Xml.Encoding.UTF_8,
                    root.getContentHandler());
        } catch (Exception ex)
        {
            throw new RuntimeException(ex);
        }

        return noticias;
    }

    private InputStream getInputStream()
    {
        try
        {
            return rssUrl.openConnection().getInputStream();
        } catch (IOException e)
        {
            throw new RuntimeException(e);
        }
    }
}
```

Debemos atender principalmente al método `parse()`. En el modelo SAX clásico nos limitamos a instanciar al handler definido en una clase independiente y llamar al correspondiente método `parse()` de SAX. Por el contrario, en este nuevo modelo SAX simplificado de Android, las acciones a realizar para cada evento las vamos a definir en esta misma clase y además asociadas a etiquetas concretas del XML. Y para ello lo primero que haremos será navegar por la estructura del XML hasta llegar a las etiquetas que nos interesa tratar y una vez allí, asignarle algunos de los *listeners* disponibles [de apertura (`StartElementListener`) o cierre (`EndElementListener`) de etiqueta] incluyendo las acciones oportunas. De esta forma, para el elemento `<item>` navegaremos hasta él obteniendo en primer lugar el elemento raíz del XML (`<rss>`) declarando un nuevo objeto `RootElement` y después accederemos a su elemento hijo `<channel>` y a su vez a su elemento hijo `<item>`, utilizando en cada paso el método `getChild()`. Una vez hemos llegado a la etiqueta deseada, asignaremos los listeners necesarios, en nuestro caso uno de apertura de etiqueta y otro de cierre, donde inicializaremos la noticia actual y la añadiremos a la lista final respectivamente, de forma análoga a lo que hacíamos para el modelo SAX clásico. Para el resto de etiquetas actuaremos de la misma forma, accediendo a ellas con `getChild()` y asignado los listeners necesarios.

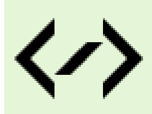
Finalmente, iniciaremos el proceso de parsing simplemente llamando al método `parse()` definido en la clase `android.Util.Xml`, al que pasaremos como parámetros el stream de entrada, la codificación del documento XML y un handler SAX obtenido directamente del objeto `RootElement` definido anteriormente.

Importante indicar, tal como vimos en el apartado anterior, que si queremos ejecutar nuestro parser sin errores en cualquier versión de Android (sobre todo a partir de la 3.0) deberemos hacerlo mediante una tarea asíncrona. Tienes más información en el apartado anterior.

Como vemos, este modelo SAX alternativo simplifica la elaboración del handler necesario y puede ayudar a evitar posibles errores en el handler y disminuir la complejidad del mismo para casos en los que el documento XML no sea tan sencillo como el utilizado para estos ejemplos. Por supuesto, el modelo clásico es tan válido

y eficiente como éste, por lo que la elección entre ambos es cuestión de gustos.

En el siguiente apartado pasaremos ya a describir el siguiente de los métodos de lectura de XML en Android, llamado *Document Object Model* (DOM).



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-xml-sax-simp

Tratamiento de XML con DOM

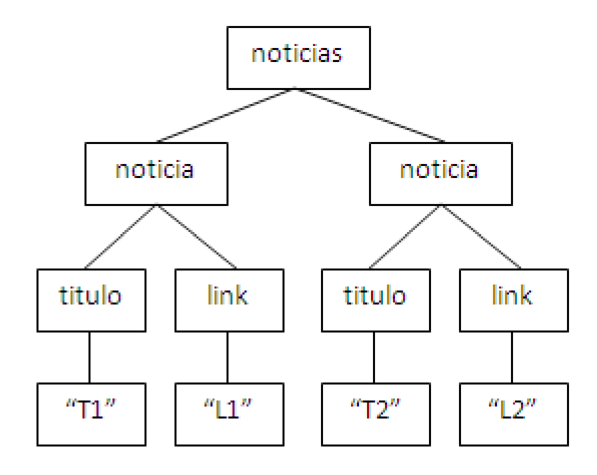
En el apartado anterior del curso de programación para Android hablamos sobre SAX, el primero de los métodos disponibles en Android para leer ficheros XML desde nuestras aplicaciones. En este segundo apartado vamos a centrarnos en DOM, otro de los métodos clásicos para la lectura y tratamiento de XML.

Cuando comentábamos la filosofía de SAX ya vimos cómo con dicho modelo el tratamiento del fichero XML se realizaba de forma secuencial, es decir, se iban realizando las acciones necesarias durante la propia lectura del documento. Sin embargo, con DOM la estrategia cambia radicalmente. Con DOM, el documento XML se lee completamente antes de poder realizar ninguna acción en función de su contenido. Esto es posible gracias a que, como resultado de la lectura del documento, el parser DOM devuelve todo su contenido en forma de una estructura de tipo árbol, donde los distintos elementos del XML se representa en forma de nodos y su jerarquía padre-hijo se establece mediante relaciones entre dichos nodos.

Como ejemplo, vemos un ejemplo de XML sencillo y cómo quedaría su representación en forma de árbol:

```
<noticias>
    <noticia>
        <titulo>T1</titulo>
        <link>L1</link>
    </noticia>
    <noticia>
        <titulo>T2</titulo>
        <link>L2</link>
    </noticia>
<noticias>
```

Este XML se traduciría en un árbol parecido al siguiente:



Como vemos, este árbol conserva la misma información contenida en el fichero XML pero en forma de nodos y transiciones entre nodos, de forma que se puede navegar fácilmente por la estructura. Además, este árbol se conserva persistente en memoria una vez leído el documento completo, lo que permite procesarlo en cualquier orden y tantas veces como sea necesario (a diferencia de SAX, donde el tratamiento era secuencial y siempre de principio a fin del documento, no pudiendo volver atrás una vez finalizada la lectura del XML).

Para todo esto, el modelo DOM ofrece una serie de clases y métodos que permiten almacenar la información de la forma descrita y facilitan la navegación y el tratamiento de la estructura creada.

Veamos cómo quedaría nuestro parser utilizando el modelo DOM y justo después comentaremos los detalles más importantes.

```
public class RssParserDom
{
    private URL rssUrl;

    public RssParserDom(String url)
    {
        try
        {
            this.rssUrl = new URL(url);
        } catch (MalformedURLException e)
        {
            throw new RuntimeException(e);
        }
    }

    public List<Noticia> parse()
    {
        //Instanciamos la fábrica para DOM
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        List<Noticia> noticias = new ArrayList<Noticia>();

        try
        {
            //Creamos un nuevo parser DOM
            DocumentBuilder builder = factory.newDocumentBuilder();

            //Realizamos lalectura completa del XML
            Document dom = builder.parse(this.getInputStream());

            //Nos posicionamos en el nodo principal del árbol (<rss>)
            Element root = dom.getDocumentElement();

            //Localizamos todos los elementos <item>
            NodeList items = root.getElementsByTagName("item");

            //Recorremos la lista de noticias
            for (int i=0; i<items.getLength(); i++)
            {
                Noticia noticia = new Noticia();

                //Obtenemos la noticia actual
                Node item = items.item(i);

                //Obtenemos la lista de datos de la noticia actual
                NodeList datosNoticia = item.getChildNodes();
```

```
                //Procesamos cada dato de la noticia
                for (int j=0; j<datosNoticia.getLength(); j++)
                {
                    Node dato = datosNoticia.item(j);
                    String etiqueta = dato.getNodeName();

                    if (etiqueta.equals("title"))
                    {
                        String texto = obtenerTexto(dato);
                        noticia.setTitulo(texto);
                    }
                    else if (etiqueta.equals("link"))
                    {
                        noticia.setLink(dato.getFirstChild().getNodeValue());
                    }
                    else if (etiqueta.equals("description"))
                    {
                        String texto = obtenerTexto(dato);
                        noticia.setDescripcion(texto);
                    }
                    else if (etiqueta.equals("guid"))
                    {
                        noticia.setGuid(dato.getFirstChild().getNodeValue());
                    }
                    else if (etiqueta.equals("pubDate"))
                    {
                        noticia.setFecha(dato.getFirstChild().getNodeValue());
                    }
                }

                noticias.add(noticia);
            }
        }
        catch (Exception ex)
        {
            throw new RuntimeException(ex);
        }

        return noticias;
    }


    private String obtenerTexto(Node dato)
    {
        StringBuilder texto = new StringBuilder();
        NodeList fragmentos = dato.getChildNodes();

        for (int k=0;k<fragmentos.getLength();k++)
        {
            texto.append(fragmentos.item(k).getNodeValue());
        }

        return texto.toString();
    }

    private InputStream getInputStream() {
        try {
            return rssUrl.openConnection().getInputStream();
        }
```

```
        catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Nos centramos una vez más en el método `parse()`. Al igual que hacíamos para SAX, el primer paso será instanciar una nueva fábrica, esta vez de tipo `DocumentBuilderFactory`, y posteriormente crear un nuevo parser a partir de ella mediante el método `newDocumentBuilder()`.

Tras esto, ya podemos realizar la lectura del documento XML llamando al método `parse()` de nuestro parser DOM, pasándole como parámetro el *stream* de entrada del fichero. Al hacer esto, el documento XML se leerá completo y se generará la estructura de árbol equivalente, que se devolverá como un objeto de tipo `Document`. Éste será el objeto que podremos navegar para realizar el tratamiento necesario del XML.

Para ello, lo primero que haremos será acceder al nodo principal del árbol (en nuestro caso, la etiqueta `<rss>`) utilizando el método `getDocumentElement()`. Una vez posicionados en dicho nodo, vamos a buscar todos los nodos cuya etiqueta sea `<item>`. Esto lo conseguimos utilizando el método de búsqueda por nombre de etiqueta, `getElementsByTagName("nombre_de_etiqueta")`, que devolverá una lista (de tipo `NodeList`) con todos los nodos hijos del nodo actual cuya etiqueta coincida con la pasada como parámetro.

Una vez tenemos localizados todos los elementos `<item>`, que representan a cada noticia, los vamos a recorrer uno a uno para ir generando todos los objetos `Noticia` necesarios. Para cada uno de ellos, se obtendrán los nodos hijos del elemento mediante `getChildNodes()` y se recorrerán éstos obteniendo su texto y almacenándolo en el atributo correspondiente del objeto `Noticia`. Para saber a qué etiqueta corresponde cada nodo hijo utilizamos el método `getNodeName()`.

Merece la pena pararnos un poco en comentar la forma de obtener el texto contenido en un nodo. Como vimos al principio del apartado en el ejemplo gráfico de árbol DOM, el texto de un nodo determinado se almacena a su vez como nodo hijo de dicho nodo. Este nodo de texto suele ser único, por lo que la forma habitual de obtener el texto de un nodo es obtener su primer nodo hijo y de éste último obtener su valor:

```
String texto = nodo.getFirstChild().getNodeValue();
```

Sin embargo, en ocasiones, el texto contenido en el nodo viene fragmentado en varios nodos hijos, en vez de sólo uno. Esto ocurre por ejemplo cuando se utilizan en el texto entidades HTML, como por ejemplo `"` . En estas ocasiones, para obtener el texto completo hay que recorrer todos los nodos hijos e ir concatenando el texto de cada uno para formar el texto completo.

Esto es lo que hace nuestra función auxiliar `obtenerTexto()`:

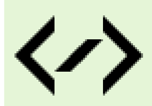
```
private String obtenerTexto(Node dato)
{
        StringBuilder texto = new StringBuilder();
        NodeList fragmentos = dato.getChildNodes();

        for (int k=0;k<fragmentos.getLength();k++)
        {
            texto.append(fragmentos.item(k).getNodeValue());
        }

        return texto.toString();
}
```

Como vemos, el modelo DOM nos permite localizar y tratar determinados elementos concretos del documento XML, sin la necesidad de recorrer todo su contenido de principio a fin. Además, a diferencia

de SAX, como tenemos cargado en memoria el documento completo de forma persistente (en forma de objeto `Document`), podremos consultar, recorrer y tratar el documento tantas veces como sea necesario sin necesidad de volverlo a parsear. En un apartado posterior veremos como todas estas características pueden ser ventajas o inconvenientes según el contexto de la aplicación y el tipo de XML tratado.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-xml-dom

Tratamiento de XML con XmlPull

En los apartados anteriores dedicados al tratamiento de XML en aplicaciones Android dentro de nuestro tutorial de programación Android hemos comentado ya los modelos SAX y DOM, los dos métodos más comunes de lectura de XML soportados en la plataforma.

En este cuarto apartado nos vamos a centrar en el último método menos conocido, aunque igual de válido según el contexto de la aplicación, llamado *XmlPull*. Este método es una versión similar al modelo StAX (*Streaming API for XML*), que en esencia es muy parecido al modelo SAX ya comentado. Y digo muy parecido porque también se basa en definir las acciones a realizar para cada uno de los eventos generados durante la lectura secuencial del documento XML. ¿Cuál es la diferencia entonces?

La diferencia radica principalmente en que, mientras que en SAX no teníamos control sobre la lectura del XML una vez iniciada (el parser lee automáticamente el XML de principio a fin generando todos los eventos necesarios), en el modelo XmlPull vamos a poder guiar o intervenir en la lectura del documento, siendo nosotros los que vayamos pidiendo de forma explícita la lectura del siguiente elemento del XML y respondiendo al resultado ejecutando las acciones oportunas.

Veamos cómo podemos hacer esto:

```
public class RssParserPull
{
    private URL rssUrl;

    public RssParserPull(String url)
    {
        try
        {
            this.rssUrl = new URL(url);
        }
        catch (MalformedURLException e)
        {
            throw new RuntimeException(e);
        }
    }

    public List<Noticia> parse()
    {
        List<Noticia> noticias = null;
        XmlPullParser parser = Xml.newPullParser();

        try
        {
            parser.setInput(this.getInputStream(), null);
```

```
            int evento = parser.getEventType();

            Noticia noticiaActual = null;

            while (evento != XmlPullParser.END_DOCUMENT)
            {
                String etiqueta = null;

                switch (evento)
                {
                    case XmlPullParser.START_DOCUMENT:

                        noticias = new ArrayList<Noticia>();
                        break;

                    case XmlPullParser.START_TAG:

                        etiqueta = parser.getName();

                        if (etiqueta.equals("item"))
                        {
                            noticiaActual = new Noticia();
                        }
                        else if (noticiaActual != null)
                        {
                            if (etiqueta.equals("link"))
                            {
                                noticiaActual.setLink(parser.nextText());
                            }
                            else if (etiqueta.equals("description"))
                            {
                                noticiaActual.setDescripcion(
                                                       parser.nextText());
                            }
                            else if (etiqueta.equals("pubDate"))
                            {
                                noticiaActual.setFecha(parser.nextText());
                            }
                            else if (etiqueta.equals("title"))
                            {
                                noticiaActual.setTitulo(parser.nextText());
                            }

                            else if (etiqueta.equals("guid"))
                            {
                                noticiaActual.setGuid(parser.nextText());
                            }
                        }
                        break;

                    case XmlPullParser.END_TAG:

                        etiqueta = parser.getName();
                        if (etiqueta.equals("item") && noticiaActual != null)
                        {
                            noticias.add(noticiaActual);
                        }
                        break;
                }
```

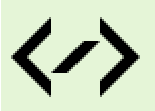
```
                evento = parser.next();
            }
        }
        catch (Exception ex)
        {
            throw new RuntimeException(ex);
        }

        return noticias;
    }

    private InputStream getInputStream()
    {
        try
        {
            return rssUrl.openConnection().getInputStream();
        }
        catch (IOException e)
        {
            throw new RuntimeException(e);
        }
    }
}
```

Centrándonos una vez más en el método `parse()`, vemos que tras crear el nuevo parser XmlPull y establecer el fichero de entrada en forma de *stream* [mediante `XmlPull.newPullParser()` y `parser.setInput(...)`] nos metemos en un bucle en el que iremos solicitando al parser en cada paso el siguiente evento encontrado en la lectura del XML, utilizando para ello el método `parser.next()`. Para cada evento devuelto como resultado consultaremos su tipo mediante el método `parser.getEventType()` y responderemos con las acciones oportunas según dicho tipo (`START_DOCUMENT`, `END_DOCUMENT`, `START_TAG`, `END_TAG`). Una vez identificado el tipo concreto de evento, podremos consultar el nombre de la etiqueta del elemento XML mediante `parser.getName()` y su texto correspondiente mediante `parser.nextText()`. En cuanto a la obtención del texto, con este modelo tenemos la ventaja de no tener que preocuparnos por "recolectar" todos los fragmentos de texto contenidos en el elemento XML, ya que `nextText()` devolverá todo el texto que encuentre hasta el próximo evento de fin de etiqueta (`ENT_TAG`).

Y sobre este modelo de tratamiento no queda mucho más que decir, ya que las acciones ejecutadas en cada caso son análogas a las que ya vimos en los apartados anteriores.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-xml-pull

Alternativas para leer/escribir XML (y otros ficheros)

En todos los ejemplos anteriores de tratamiento de XML decidí utilizar como entrada un fichero online, que obteníamos desde una determinada URL, porque pensé que sería el caso más típico que íbamos a necesitar en la mayoría de aplicaciones. Sin embargo, quiero también dedicar un capítulo breve a describir la tarea de tratar un fichero XML que se encuentre en un recurso local. Veamos varias alternativas.

Escribir ficheros XML en Android

Para escribir ficheros XML sin meternos en muchas complicaciones innecesarias se me ocurren básicamente

dos formas. La primera de ellas, y la más sencilla y directa, es construir manualmente el XML utilizando por ejemplo un objeto `StringBuilder` y tras esto escribir el resultado a un fichero en memoria interna o externa (no me detendré mucho por ahora en esto último porque dedicaremos al tema un par de capítulos más adelante). Esto quedaría de una forma similar a la siguiente:

```
//Creamos un fichero en la memoria interna
OutputStreamWriter fout =
    new OutputStreamWriter(
        openFileOutput("prueba.xml",
            Context.MODE_PRIVATE));

StringBuilder sb = new StringBuilder();

//Construimos el XML
sb.append("");
sb.append("" + "Usuario1" + "");
sb.append("" + "ApellidosUsuario1" + "");
sb.append("");

//Escribimos el resultado a un fichero
fout.write(sb.toString());
fout.close();
```

Como método alternativo también podemos utilizar el *serializador* XML incluido con la API XmlPull. Aunque no es un método tan directo como el anterior no deja de ser bastante intuitivo. Los pasos para conseguir esto serán crear el objeto `XmlSerializer`, crear el fichero de salida, asignar el fichero como salida del serializador y construir el XML mediante los métodos `startTag()`, `text()` y `endTag()` pasándoles los nombres de etiqueta y contenidos de texto correspondientes (aunque existen más métodos, por ejemplo para escribir atributos de una etiqueta, éstos tres son los principales). Finalmente deberemos llamar a `endDocument()` para finalizar y cerrar nuestro documento XML. Un ejemplo equivalente al anterior utilizando este método sería el siguiente:

```
//Creamos el serializer
XmlSerializer ser = Xml.newSerializer();

//Creamos un fichero en memoria interna
OutputStreamWriter fout =
    new OutputStreamWriter(
        openFileOutput("prueba_pull.xml",
            Context.MODE_PRIVATE));

//Asignamos el resultado del serializer al fichero
ser.setOutput(fout);

//Construimos el XML
ser.startTag("", "usuario");

ser.startTag("", "nombre");
ser.text("Usuario1");
ser.endTag("", "nombre");

ser.startTag("", "apellidos");
ser.text("ApellidosUsuario1");
ser.endTag("", "apellidos");
```

```
ser.endTag("", "usuario");

ser.endDocument();

fout.close();
```

Así de sencillo, no merece la pena complicarse la vida con otros métodos más complicados.

Leer ficheros XML en Android desde recursos locales

Para leer ficheros XML desde un recurso local se me ocurren varias posibilidades, por ejemplo leerlo desde la memoria interna/externa del dispositivo, leer un XML desde un recurso XML (carpeta `/res/xml`), desde un recurso *Raw* (carpeta `/res/raw`), o directamente desde la carpeta `/assets` de nuestro proyecto. Salvo en el segundo caso (recurso XML), en todos los demás la solución va a pasar por conseguir una referencia de tipo `InputStream` (os recuerdo que cualquiera de los métodos que vimos para leer un XML partían de una referencia de este tipo) a partir de nuestro fichero o recurso XML, sea cual sea su localización.

Así, por ejemplo, si el fichero XML está almacenado en la memoria interna de nuestro dispositivo, podríamos acceder a él mediante el método `openFileInput()` tal como vimos en el capítulo dedicado al tratamiento de ficheros. Este método devuelve un objeto de tipo `FileInputStream`, que al derivar de `InputStream` podemos utilizarlo como entrada a cualquiera de los mecanismos de lectura de XML (SAX, DOM, XmlPull).

```
//Obtenemos la referencia al fichero XML de entrada
FileInputStream fil = openFileInput("prueba.xml");

//DOM (Por ejemplo)
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();

DocumentBuilder builder = factory.newDocumentBuilder();
Document dom = builder.parse(fil);

//A partir de aquí se trataría el árbol DOM como siempre.
//Por ejemplo:
Element root = dom.getDocumentElement();

//...
```

En el caso de encontrarse el fichero como recurso Raw, es decir, en la carpeta `/res/raw`, tendríamos que obtener la referencia al fichero mediante el método `getRawResource()` pasándole como parámetro el ID de recurso del fichero. Esto nos devuelve directamente el *stream* de entrada en forma de `InputStream`.

```
//Obtenemos la referencia al fichero XML de entrada
InputStream is = getResources().openRawResource(R.raw.prueba);

//DOM (Por ejemplo)
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();

DocumentBuilder builder = factory.newDocumentBuilder();
Document dom = builder.parse(is);

//A partir de aquí se trataría el árbol DOM como siempre.
//Por ejemplo:
Element root = dom.getDocumentElement();

//...
```

Por último, si el XML se encontrara en la carpeta `/assets` de nuestra aplicación, accederíamos a él a través del `AssetManager`, el cual podemos obtenerlo con el método `getAssets()` de nuestra actividad. Sobre este `AssetManager` tan sólo tendremos que llamar al método `open()` con el nombre del fichero para obtener una referencia de tipo `InputStream` a nuestro XML.

```
//Obtenemos la referencia al fichero XML de entrada
AssetManager assetMan = getAssets();
InputStream is = assetMan.open("prueba_asset.xml");

//DOM (Por ejemplo)
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();

DocumentBuilder builder = factory.newDocumentBuilder();
Document dom = builder.parse(is);

//A partir de aquí se trataría el árbol DOM como siempre.
//Por ejemplo:
Element root = dom.getDocumentElement();

//...
```

El último caso, algo distinto a los anteriores, pasaría por leer el XML desde un recurso XML localizado en la carpeta `/res/xml` de nuestro proyecto. Para acceder a un recurso de este tipo debemos utilizar el método `getXml()` al cual debemos pasarle como parámetro el ID de recurso del fichero XML. Esto nos devolverá un objeto `XmlResourceParser`, que no es más que un parser de tipo XmlPull como el que ya comentamos en su momento. Por tanto, la forma de trabajar con este parser será idéntica a la que ya conocemos, por ejemplo:

```
//Obtenemos un parser XmlPull asociado a nuestro XML
XmlResourceParser xrp = getResources().getXml(R.xml.prueba);

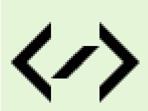
//A partir de aquí utilizamos la variable 'xrp' como
//cualquier otro parser de tipo XmlPullParser. Por ejemplo:

int evento = xrp.getEventType();

if(evento == XmlPullParser.START_DOCUMENT)
    Log.i("XmlTips", "Inicio del documento");

//...
```

Por último, indicar que todas estas formas de acceder a un fichero en Android no se limitan únicamente a los de tipo XML, sino que pueden utilizarse para leer cualquier otro tipo de ficheros.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-xml-tips

6

Bases de Datos

VI. Bases de Datos

Primeros pasos con SQLite

En los siguientes apartados de este manual, nos vamos a detener en describir las distintas opciones de acceso a datos que proporciona la plataforma y en cómo podemos realizar las tareas más habituales.

La plataforma Android proporciona dos herramientas principales para el almacenamiento y consulta de datos estructurados:

- Bases de Datos SQLite
- Content Providers

En estos próximos apartados nos centraremos en la primera opción, SQLite, que abarcará todas las tareas relacionadas con el almacenamiento de los datos propios de nuestra aplicación. El segundo de los mecanismos, los *Content Providers*, que trataremos más adelante, nos facilitarán la tarea de hacer visibles esos datos a otras aplicaciones y, de forma recíproca, de permitir la consulta de datos publicados por terceros desde nuestra aplicación.

SQLite es un motor de bases de datos muy popular en la actualidad por ofrecer características tan interesantes como su pequeño tamaño, no necesitar servidor, precisar poca configuración, ser transaccional, y por supuesto ser de código libre.

Android incorpora de serie todas las herramientas necesarias para la creación y gestión de bases de datos SQLite, y entre ellas una completa API para llevar a cabo de manera sencilla todas las tareas necesarias. Sin embargo, en este primer apartado sobre bases de datos en Android no vamos a entrar en mucho detalle con esta API. Por el momento nos limitaremos a ver el código necesario para crear una base de datos, insertaremos algún dato de prueba, y veremos cómo podemos comprobar que todo funciona correctamente.

En Android, la forma típica para crear, actualizar, y conectar con una base de datos SQLite será a través de una clase auxiliar llamada `SQLiteOpenHelper`, o para ser más exactos, de una clase propia que derive de ella y que debemos personalizar para adaptarnos a las necesidades concretas de nuestra aplicación.

La clase `SQLiteOpenHelper` tiene tan sólo un constructor, que normalmente no necesitaremos sobrescribir, y dos métodos abstractos, `onCreate()` y `onUpgrade()`, que deberemos personalizar con el código necesario para crear nuestra base de datos y para actualizar su estructura respectivamente.

Como ejemplo, nosotros vamos a crear una base de datos muy sencilla llamada `BDUsuarios`, con una sola tabla llamada *Usuarios* que contendrá sólo dos campos: *nombre* e *email*. Para ello, vamos a crear una clase derivada de `SQLiteOpenHelper` que llamaremos `UsuariosSQLiteHelper`, donde sobrescribiremos los métodos `onCreate()` y `onUpgrade()` para adaptarlos a la estructura de datos indicada:

```
package net.sgoliver.android.bd;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.database.sqlite.SQLiteOpenHelper;

public class UsuariosSQLiteHelper extends SQLiteOpenHelper {

    //Sentencia SQL para crear la tabla de Usuarios
    String sqlCreate = "CREATE TABLE Usuarios (codigo INTEGER, nombre TEXT)";
```

```
    public UsuariosSQLiteHelper(Context contexto, String nombre,
                                CursorFactory factory, int version) {
        super(contexto, nombre, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        //Se ejecuta la sentencia SQL de creación de la tabla
        db.execSQL(sqlCreate);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int versionAnterior,
                          int versionNueva) {
        //NOTA: Por simplicidad del ejemplo aquí utilizamos directamente
        //      la opción de eliminar la tabla anterior y crearla de nuevo
        //      vacía con el nuevo formato.
        //      Sin embargo lo normal será que haya que migrar datos de la
        //      tabla antigua a la nueva, por lo que este método debería
        //      ser más elaborado.

        //Se elimina la versión anterior de la tabla
        db.execSQL("DROP TABLE IF EXISTS Usuarios");

        //Se crea la nueva versión de la tabla
        db.execSQL(sqlCreate);
    }
}
```

Lo primero que hacemos es definir una variable llamado `sqlCreate` donde almacenamos la sentencia SQL para crear una tabla llamada *Usuarios* con los campos alfanuméricos nombre e email. NOTA: No es objetivo de este tutorial describir la sintaxis del lenguaje SQL ni las particularidades del motor de base de datos SQLite, por lo que no entraré a describir las sentencias SQL utilizadas. Para más información sobre SQLite puedes consultar la documentación oficial o empezar por leer una pequeña introducción que hice en mi blog cuando traté el tema de utilizar SQLite desde aplicaciones .NET

El método `onCreate()` será ejecutado automáticamente por nuestra clase `UsuariosDBHelper` cuando sea necesaria la creación de la base de datos, es decir, cuando aún no exista. Las tareas típicas que deben hacerse en este método serán la creación de todas las tablas necesarias y la inserción de los datos iniciales si son necesarios. En nuestro caso, sólo vamos a crear la tabla *Usuarios* descrita anteriormente. Para la creación de la tabla utilizaremos la sentencia SQL ya definida y la ejecutaremos contra la base de datos utilizando el método más sencillo de los disponibles en la API de SQLite proporcionada por Android, llamado `execSQL()`. Este método se limita a ejecutar directamente el código SQL que le pasemos como parámetro.

Por su parte, el método `onUpgrade()` se lanzará automáticamente cuando sea necesaria una actualización de la estructura de la base de datos o una conversión de los datos.

Un ejemplo práctico: imaginemos que publicamos una aplicación que utiliza una tabla con los campos *usuario* e *email* (llamémoslo versión 1 de la base de datos). Más adelante, ampliamos la funcionalidad de nuestra aplicación y necesitamos que la tabla también incluya un campo adicional como por ejemplo con la *edad* del usuario (versión 2 de nuestra base de datos). Pues bien, para que todo funcione correctamente, la primera vez que ejecutemos la versión ampliada de la aplicación necesitaremos modificar la estructura de la tabla Usuarios para añadir el nuevo campo *edad*. Pues este tipo de cosas son las que se encargará de hacer automáticamente el método `onUpgrade()` cuando intentemos abrir una versión concreta de la base de datos que aún no exista. Para ello, como parámetros recibe la versión actual de la base de datos en el sistema, y la nueva versión a la que se quiere convertir. En función de esta pareja de datos necesitaremos realizar unas acciones u otras. En nuestro caso de ejemplo optamos por la opción más sencilla: borrar la

tabla actual y volver a crearla con la nueva estructura, pero como se indica en los comentarios del código, lo habitual será que necesitemos algo más de lógica para convertir la base de datos de una versión a otra y por supuesto para conservar los datos registrados hasta el momento.

Una vez definida nuestra clase *helper*, la apertura de la base de datos desde nuestra aplicación resulta ser algo de lo más sencillo. Lo primero será crear un objeto de la clase `UsuariosSQLiteHelper` al que pasaremos el contexto de la aplicación (en el ejemplo una referencia a la actividad principal), el nombre de la base de datos, un objeto `CursorFactory` que típicamente no será necesario (en ese caso pasaremos el valor `null`), y por último la versión de la base de datos que necesitamos. La simple creación de este objeto puede tener varios efectos:

- Si la base de datos ya existe y su versión actual coincide con la solicitada simplemente se realizará la conexión con ella.
- Si la base de datos existe pero su versión actual es anterior a la solicitada, se llamará automáticamente al método `onUpgrade()` para convertir la base de datos a la nueva versión y se conectará con la base de datos convertida.
- Si la base de datos no existe, se llamará automáticamente al método `onCreate()` para crearla y se conectará con la base de datos creada.

Una vez tenemos una referencia al objeto `UsuariosSQLiteHelper`, llamaremos a su método `getReadableDatabase()` o `getWritableDatabase()` para obtener una referencia a la base de datos, dependiendo si sólo necesitamos consultar los datos o también necesitamos realizar modificaciones, respectivamente.

Ahora que ya hemos conseguido una referencia a la base de datos (objeto de tipo `SQLiteDatabase`) ya podemos realizar todas las acciones que queramos sobre ella. Para nuestro ejemplo nos limitaremos a insertar 5 registros de prueba, utilizando para ello el método ya comentado `execSQL()` con las sentencias `INSERT` correspondientes. Por último cerramos la conexión con la base de datos llamando al método `close()`.

```
package net.sgoliver.android.bd;

import android.app.Activity;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;

public class AndroidBaseDatos extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        //Abrimos la base de datos 'DBUsuarios' en modo escritura
        UsuariosSQLiteHelper usdbh =
            new UsuariosSQLiteHelper(this, "DBUsuarios", null, 1);

        SQLiteDatabase db = usdbh.getWritableDatabase();

        //Si hemos abierto correctamente la base de datos
        if(db != null)
        {
            //Insertamos 5 usuarios de ejemplo
            for(int i=1; i<=5; i++)
            {
```

```
                //Generamos los datos
                int codigo = i;
                String nombre = "Usuario" + i;

                //Insertamos los datos en la tabla Usuarios
                db.execSQL("INSERT INTO Usuarios (codigo, nombre) " +
                           "VALUES (" + codigo + ", '" + nombre +"')");
            }

            //Cerramos la base de datos
            db.close();
        }
    }
}
```

Vale, ¿y ahora qué? ¿dónde está la base de datos que acabamos de crear? ¿cómo podemos comprobar que todo ha ido bien y que los registros se han insertado correctamente? Vayamos por partes.

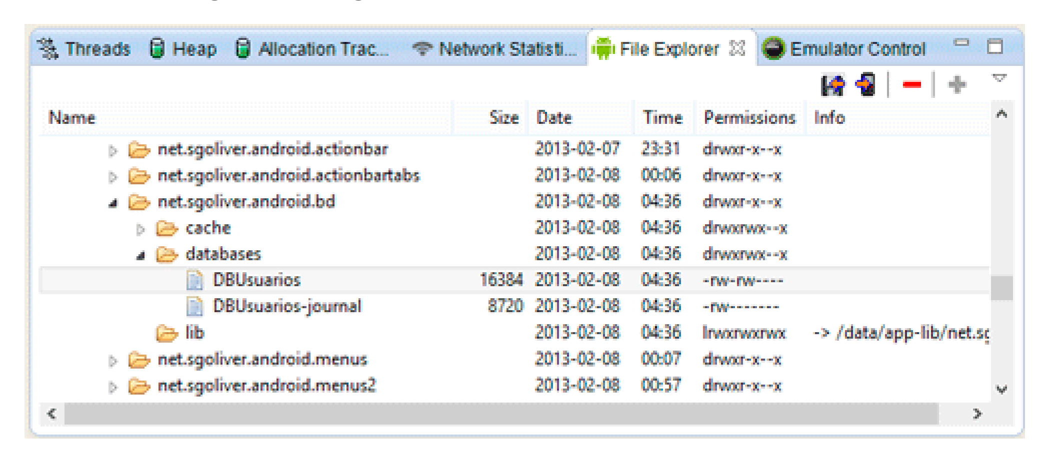
En primer lugar veamos dónde se ha creado nuestra base de datos. Todas las bases de datos SQLite creadas por aplicaciones Android utilizando este método se almacenan en la memoria del teléfono en un fichero con el mismo nombre de la base de datos situado en una ruta que sigue el siguiente patrón:

`/data/data/paquete.java.de.la.aplicacion/databases/nombre_base_datos`

En el caso de nuestro ejemplo, la base de datos se almacenaría por tanto en la ruta siguiente:

`/data/data/net.sgoliver.android.bd/databases/DBUsuarios`

Para comprobar esto podemos hacer lo siguiente. Una vez ejecutada por primera vez desde Eclipse la aplicación de ejemplo sobre el emulador de Android (y por supuesto antes de cerrarlo) podemos ir a la perspectiva "DDMS" (*Dalvik Debug Monitor Server*) de Eclipse y en la solapa "*File Explorer*" podremos acceder al sistema de archivos del emulador, donde podremos buscar la ruta indicada de la base de datos. Podemos ver esto en la siguiente imagen:



Con esto ya comprobamos al menos que el fichero de nuestra base de datos se ha creado en la ruta correcta. Ya sólo nos queda comprobar que tanto las tablas creadas como los datos insertados también se han incluido correctamente en la base de datos. Para ello podemos recurrir a dos posibles métodos:

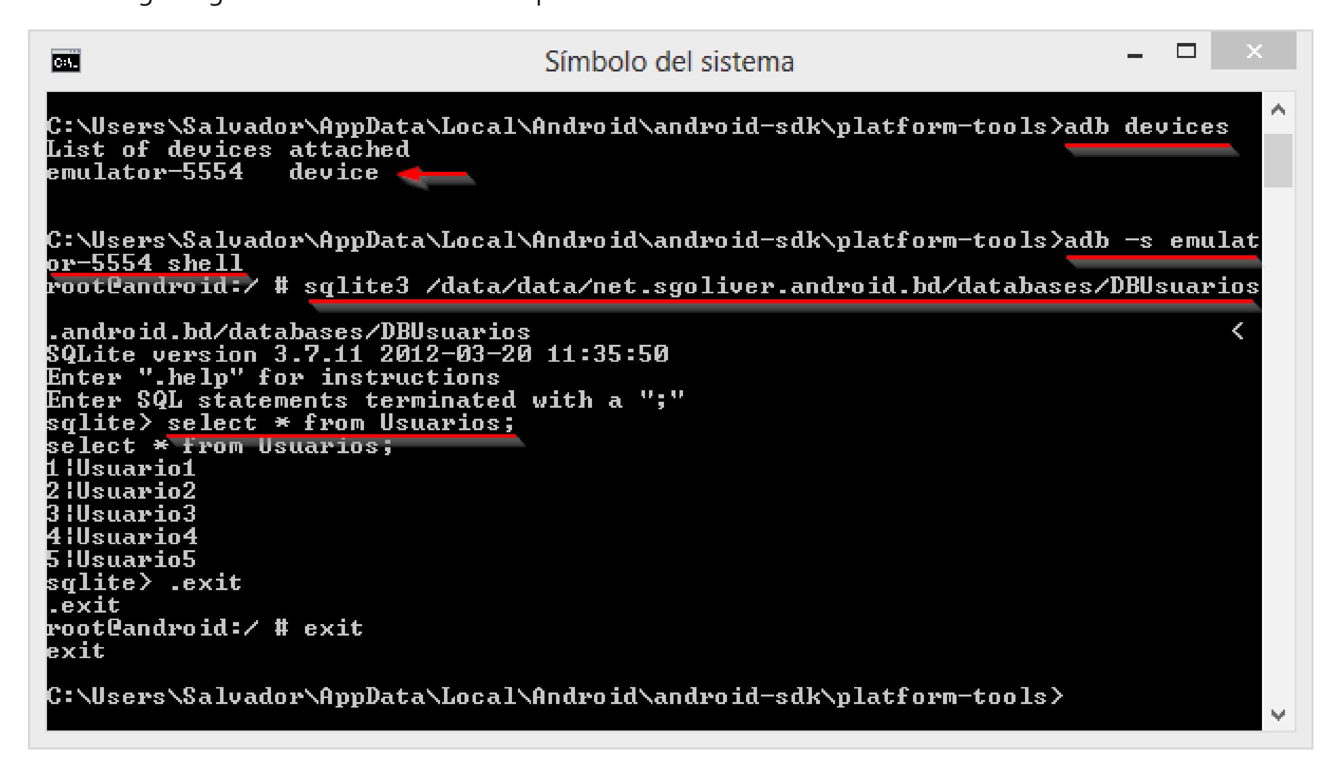
1. Transferir la base de datos a nuestro PC y consultarla con cualquier administrador de bases de datos SQLite.
2. Acceder directamente a la consola de comandos del emulador de Android y utilizar los comandos existentes para acceder y consultar la base de datos SQLite.

El primero de los métodos es sencillo. El fichero de la base de datos podemos transferirlo a nuestro PC

utilizando el botón de descarga situado en la esquina superior derecha del explorador de archivos (remarcado en rojo en la imagen anterior). Junto a este botón aparecen otros dos para hacer la operación contraria (copiar un fichero local al sistema de archivos del emulador) y para eliminar ficheros del emulador. Una vez descargado el fichero a nuestro sistema local, podemos utilizar cualquier administrador de SQLite para abrir y consultar la base de datos, por ejemplo SQLite Administrator (freeware).

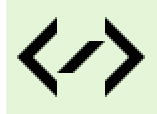
El segundo método utiliza una estrategia diferente. En vez de descargar la base de datos a nuestro sistema local, somos nosotros los que accedemos de forma remota al emulador a través de su consola de comandos (*shell*). Para ello, con el emulador de Android aún abierto, debemos abrir una consola de MS-DOS y utilizar la utilidad adb.exe (*Android Debug Bridge*) situada en la carpeta `platform-tools` del SDK de Android (en mi caso: `c:\Users\Salvador\AppData\Local\Android\android-sdk\platform-tools\`). En primer lugar consultaremos los identificadores de todos los emuladores en ejecución mediante el comando "`adb devices`". Esto nos debe devolver una única instancia si sólo tenemos un emulador abierto, que en mi caso particular se llama "`emulator-5554`".

Tras conocer el identificador de nuestro emulador, vamos a acceder a su shell mediante el comando "`adb -s identificador-del-emulador shell`". Una vez conectados, ya podemos acceder a nuestra base de datos utilizando el comando `sqlite3` pasándole la ruta del fichero, para nuestro ejemplo "`sqlite3 /data/data/net.sgoliver.android.bd/ databases/DBUsuarios`". Si todo ha ido bien, debe aparecernos el *prompt* de SQLite "`sqlite>`", lo que nos indicará que ya podemos escribir las consultas SQL necesarias sobre nuestra base de datos. Nosotros vamos a comprobar que existe la tabla Usuarios y que se han insertado los cinco registros de ejemplo. Para ello haremos la siguiente consulta: "`SELECT * FROM Usuarios;`". Si todo es correcto esta instrucción debe devolvernos los cinco usuarios existentes en la tabla. En la imagen siguiente se muestra todo el proceso descrito:



Con esto ya hemos comprobado que nuestra base de datos se ha creado correctamente, que se han insertado todos los registros de ejemplo y que todo funciona según se espera.

En los siguientes apartados comentaremos las distintas posibilidades que tenemos a la hora de manipular los datos de la base de datos (insertar, eliminar y modificar datos) y cómo podemos realizar consultas sobre los mismos, ya que [como siempre] tendremos varias opciones disponibles.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-basedatos-1

Insertar/Actualizar/Eliminar registros de la BD

En el apartado anterior del curso de programación en Android vimos cómo crear una base de datos para utilizarla desde nuestra aplicación Android. En este segundo apartado de la serie vamos a describir las posibles alternativas que proporciona la API de Android a la hora de insertar, actualizar y eliminar registros de nuestra base de datos SQLite.

La API de SQLite de Android proporciona dos alternativas para realizar operaciones sobre la base de datos que no devuelven resultados (entre ellas la inserción/actualización/eliminación de registros, pero también la creación de tablas, de índices, etc).

El primero de ellos, que ya comentamos brevemente en el apartado anterior, es el método `execSQL()` de la clase `SQLiteDatabase`. Este método permite ejecutar cualquier sentencia SQL sobre la base de datos, siempre que ésta no devuelva resultados. Para ello, simplemente aportaremos como parámetro de entrada de este método la cadena de texto correspondiente con la sentencia SQL. Cuando creamos la base de datos en el post anterior ya vimos algún ejemplo de esto para insertar los registros de prueba. Otros ejemplos podrían ser los siguientes:

```
//Insertar un registro
db.execSQL("INSERT INTO Usuarios (codigo,nombre) VALUES ('6','usuariopru')");

//Eliminar un registro
db.execSQL("DELETE FROM Usuarios WHERE codigo=6");

//Actualizar un registro
db.execSQL("UPDATE Usuarios SET nombre='usunuevo' WHERE codigo=6");
```

La segunda de las alternativas disponibles en la API de Android es utilizar los métodos `insert()`, `update()` y `delete()` proporcionados también con la clase `SQLiteDatabase`. Estos métodos permiten realizar las tareas de inserción, actualización y eliminación de registros de una forma algo más paramétrica que `execSQL()`, separando tablas, valores y condiciones en parámetros independientes de estos métodos.

Empecemos por el método `insert()` para insertar nuevos registros en la base de datos. Este método recibe tres parámetros, el primero de ellos será el nombre de la tabla, el tercero serán los valores del registro a insertar, y el segundo lo obviaremos por el momento ya que tan sólo se hace necesario en casos muy puntuales (por ejemplo para poder insertar registros completamente vacíos), en cualquier otro caso pasaremos con valor `null` este segundo parámetro.

Los valores a insertar los pasaremos como elementos de una colección de tipo `ContentValues`. Esta colección es de tipo diccionario, donde almacenaremos parejas de clave-valor, donde la clave será el nombre de cada campo y el valor será el dato correspondiente a insertar en dicho campo. Veamos un ejemplo:

```
//Creamos el registro a insertar como objeto ContentValues
ContentValues nuevoRegistro = new ContentValues();
nuevoRegistro.put("codigo", "6");
nuevoRegistro.put("nombre","usuariopru");

//Insertamos el registro en la base de datos
db.insert("Usuarios", null, nuevoRegistro);
```

Los métodos `update()` y `delete()` se utilizarán de forma muy parecida a ésta, con la salvedad de que recibirán un parámetro adicional con la condición `WHERE` de la sentencia SQL. Por ejemplo, para actualizar el nombre del usuario con código '6' haríamos lo siguiente:

```
//Establecemos los campos-valores a actualizar
ContentValues valores = new ContentValues();
valores.put("nombre","usunuevo");

//Actualizamos el registro en la base de datos
db.update("Usuarios", valores, "codigo=6", null);
```

Como podemos ver, como tercer parámetro del método `update()` pasamos directamente la condición del `UPDATE` tal como lo haríamos en la cláusula `WHERE` en una sentencia SQL normal.

El método `delete()` se utilizaría de forma análoga. Por ejemplo para eliminar el registro del usuario con código '6' haríamos lo siguiente:

```
//Eliminamos el registro del usuario '6'
db.delete("Usuarios", "codigo=6", null);
```

Como vemos, volvemos a pasar como primer parámetro el nombre de la tabla y en segundo lugar la condición `WHERE`. Por supuesto, si no necesitáramos ninguna condición, podríamos dejar como `null` en este parámetro (lo que eliminaría todos los registros de la tabla).

Un último detalle sobre estos métodos. Tanto en el caso de `execSQL()` como en los casos de `update()` o `delete()` podemos utilizar argumentos dentro de las condiciones de la sentencia SQL. Éstos no son más que partes variables de la sentencia SQL que aportaremos en un *array* de valores aparte, lo que nos evitará pasar por la situación típica en la que tenemos que construir una sentencia SQL concatenando cadenas de texto y variables para formar el comando SQL final. Estos argumentos SQL se indicarán con el símbolo '`?`', y los valores de dichos argumentos deben pasarse en el array en el mismo orden que aparecen en la sentencia SQL. Así, por ejemplo, podemos escribir instrucciones como la siguiente:

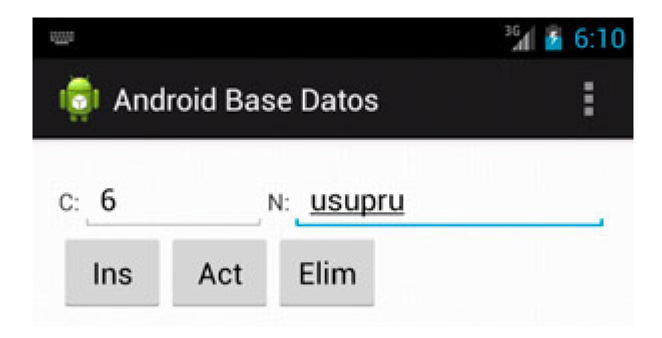
```
//Eliminar un registro con execSQL(), utilizando argumentos
String[] args = new String[]{"usuario1"};
db.execSQL("DELETE FROM Usuarios WHERE nombre=?", args);

//Actualizar dos registros con update(), utilizando argumentos
ContentValues valores = new ContentValues();
valores.put("nombre","usunuevo");

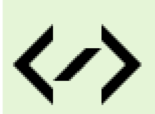
String[] args = new String[]{"usuario1", "usuario2"};
db.update("Usuarios", valores, "nombre=? OR nombre=?", args);
```

Esta forma de pasar a la sentencia SQL determinados datos variables puede ayudarnos además a escribir código más limpio y evitar posibles errores.

Para este apartado he continuado con la aplicación de ejemplo del apartado anterior, a la que he añadido dos cuadros de texto para poder introducir el código y nombre de un usuario y tres botones para insertar, actualizar o eliminar dicha información.



En el siguiente apartado veremos cómo consultar la base de datos para recuperar registros según un determinado criterio.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-basedatos-2

Consultar/Recuperar registros de la BD

En el anterior apartado vimos todas las opciones disponibles a la hora de insertar, actualizar y eliminar datos de una base de datos SQLite en Android. En esta nueva entrega vamos a describir la última de las tareas importantes de tratamiento de datos que nos queda por ver, la selección y recuperación de datos.

De forma análoga a lo que vimos para las sentencias de modificación de datos, vamos a tener dos opciones principales para recuperar registros de una base de datos SQLite en Android. La primera de ellas utilizando directamente un comando de selección SQL, y como segunda opción utilizando un método específico donde parametrizaremos la consulta a la base de datos.

Para la primera opción utilizaremos el método `rawQuery()` de la clase `SQLiteDatabase`. Este método recibe directamente como parámetro un comando SQL completo, donde indicamos los campos a recuperar y los criterios de selección. El resultado de la consulta lo obtendremos en forma de cursor, que posteriormente podremos recorrer para procesar los registros recuperados. Sirva la siguiente consulta a modo de ejemplo:

```
Cursor c = db.rawQuery(" SELECT codigo,nombre FROM Usuarios WHERE
nombre='usu1' ", null);
```

Como en el caso de los métodos de modificación de datos, también podemos añadir a este método una lista de argumentos variables que hayamos indicado en el comando SQL con el símbolo '`?`', por ejemplo así:

```
String[] args = new String[] {"usu1"};
Cursor c = db.rawQuery(" SELECT codigo,nombre FROM Usuarios WHERE nombre=? ",
args);
```

Más adelante en este apartado veremos cómo podemos manipular el objeto `Cursor` para recuperar los datos obtenidos.

Como segunda opción para recuperar datos podemos utilizar el método `query()` de la clase `SQLiteDatabase`. Este método recibe varios parámetros: el nombre de la tabla, un array con los nombre de campos a recuperar, la cláusula `WHERE`, un array con los argumentos variables incluidos en el `WHERE` (si los hay, `null` en caso contrario), la cláusula GROUP BY si existe, la cláusula `HAVING` si existe, y por último la cláusula `ORDER BY` si existe. Opcionalmente, se puede incluir un parámetro al final más indicando el número máximo de registros que queremos que nos devuelva la consulta. Veamos el mismo ejemplo anterior utilizando el método `query()`:

```
String[] campos = new String[] {"codigo", "nombre"};
String[] args = new String[] {"usu1"};

Cursor c = db.query("Usuarios", campos, "usuario=?", args, null, null, null);
```

Como vemos, los resultados se devuelven nuevamente en un objeto `Cursor` que deberemos recorrer para procesar los datos obtenidos.

Para recorrer y manipular el cursor devuelto por cualquiera de los dos métodos mencionados tenemos a nuestra disposición varios métodos de la clase `Cursor`, entre los que destacamos dos de los dedicados a recorrer el cursor de forma secuencial y en orden natural:

- `moveToFirst()`: mueve el puntero del cursor al primer registro devuelto.
- `moveToNext()`: mueve el puntero del cursor al siguiente registro devuelto.

Los métodos `moveToFirst()` y `moveToNext()` devuelven `TRUE` en caso de haber realizado el movimiento correspondiente del puntero sin errores, es decir, siempre que exista un primer registro o un registro siguiente, respectivamente.

Una vez posicionados en cada registro podremos utilizar cualquiera de los métodos `getXXX(índice_columna)` existentes para cada tipo de dato para recuperar el dato de cada campo del registro actual del cursor. Así, si queremos recuperar por ejemplo la segunda columna del registro actual, y ésta contiene un campo alfanumérico, haremos la llamada `getString(1)` [NOTA: los índices comienzan por 0 (cero), por lo que la segunda columna tiene índice 1], en caso de contener un dato de tipo real llamaríamos a `getDouble(1)`, y de forma análoga para todos los tipos de datos existentes. Con todo esto en cuenta, veamos cómo podríamos recorrer el cursor devuelto por el ejemplo anterior:

```
String[] campos = new String[] {"codigo", "nombre"};
String[] args = new String[] {"usu1"};

Cursor c = db.query("Usuarios", campos, "nombre=?", args, null, null, null);

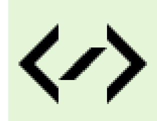
//Nos aseguramos de que existe al menos un registro
if (c.moveToFirst()) {
     //Recorremos el cursor hasta que no haya más registros
     do {
          String codigo = c.getString(0);
          String nombre = c.getString(1);
     } while(c.moveToNext());
}
```

Además de los métodos comentados de la clase `Cursor` existen muchos más que nos pueden ser útiles en muchas ocasiones. Por ejemplo, `getCount()` te dirá el número total de registros devueltos en el cursor, `getColumnName(i)` devuelve el nombre de la columna con índice i, `moveToPosition(i)` mueve el puntero del cursor al registro con índice i, etc. Podéis consultar la lista completa de métodos disponibles en la clase Cursor en la documentación oficial de Android.

En este apartado he seguido ampliando la aplicación de ejemplo anterior para añadir la posibilidad de recuperar todos los registros de la tabla Usuarios pulsando un nuevo botón de consulta.



Con esto, terminamos la serie de apartados básicos dedicados a las tareas de mantenimiento de datos en aplicaciones Android mediante bases de datos SQLite. Soy consciente de que dejamos en el tintero algunos temas algo más avanzados (como por ejemplo el uso de *transacciones*, que intentaré tratar más adelante), pero con los métodos descritos podremos realizar un porcentaje bastante alto de todas las tareas necesarias relativas al tratamiento de datos estructurados en aplicaciones Android.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-basedatos-3

7

Preferencias

VII. Preferencias en Android

Preferencias Compartidas

Las preferencias no son más que datos que una aplicación debe guardar de algún modo para personalizar la experiencia del usuario, por ejemplo información personal, opciones de presentación, etc. En apartados anteriores vimos ya uno de los métodos disponibles en la plataforma Android para almacenar datos, como son las bases de datos SQLite. Las preferencias de una aplicación se podrían almacenar por su puesto utilizando este método, y no tendría nada de malo, pero Android proporciona otro método alternativo diseñado específicamente para administrar este tipo de datos: las *preferencias compartidas* o *shared preferences*. Cada preferencia se almacenará en forma de clave-valor, es decir, cada una de ellas estará compuesta por un identificador único (p.e. "email") y un valor asociado a dicho identificador (p.e. "prueba@email.com"). Además, y a diferencia de SQLite, los datos no se guardan en un fichero binario de base de datos, sino en ficheros XML como veremos al final de este apartado.

La API para el manejo de estas preferencias es muy sencilla. Toda la gestión se centraliza en la clase `SharedPrefences`, que representará a una colección de preferencias. Una aplicación Android puede gestionar varias colecciones de preferencias, que se diferenciarán mediante un identificador único. Para obtener una referencia a una colección determinada utilizaremos el método `getSharedPrefences()` al que pasaremos el identificador de la colección y un modo de acceso. El modo de acceso indicará qué aplicaciones tendrán acceso a la colección de preferencias y qué operaciones tendrán permitido realizar sobre ellas. Así, tendremos tres posibilidades principales:

- `MODE_PRIVATE`. Sólo nuestra aplicación tiene acceso a estas preferencias.
- `MODE_WORLD_READABLE`. Todas las aplicaciones pueden leer estas preferencias, pero sólo la nuestra puede modificarlas.
- `MODE_WORLD_WRITABLE`. Todas las aplicaciones pueden leer y modificar estas preferencias.

Las dos últimas opciones son relativamente "peligrosas" por lo que en condiciones normales no deberían usarse. De hecho, se han declarado como obsoletas en la API 17 (Android 4.2).

Teniendo todo esto en cuenta, para obtener una referencia a una colección de preferencias llamada por ejemplo "*MisPreferencias*" y como modo de acceso exclusivo para nuestra aplicación haríamos lo siguiente:

```
SharedPreferences prefs =
     getSharedPreferences("MisPreferencias", Context.MODE_PRIVATE);
```

Una vez hemos obtenido una referencia a nuestra colección de preferencias, ya podemos obtener, insertar o modificar preferencias utilizando los métodos `get` o `put` correspondientes al tipo de dato de cada preferencia. Así, por ejemplo, para obtener el valor de una preferencia llamada "email" de tipo `String` escribiríamos lo siguiente:

```
SharedPreferences prefs =
     getSharedPreferences("MisPreferencias",Context.MODE_PRIVATE);

String correo = prefs.getString("email", "por_defecto@email.com");
```

Como vemos, al método `getString()` le pasamos el nombre de la preferencia que queremos recuperar y un segundo parámetro con un valor por defecto. Este valor por defecto será el devuelto por el método `getString()` si la preferencia solicitada no existe en la colección. Además del método `getString()`, existen por supuesto métodos análogos para el resto de tipos de datos básicos, por ejemplo `getInt()`, getLong(), `getFloat()`, `getBoolean()`, …

Para actualizar o insertar nuevas preferencias el proceso será igual de sencillo, con la única diferencia de que la actualización o inserción no la haremos directamente sobre el objeto `SharedPreferences`, sino sobre su objeto de edición `SharedPreferences.Editor`. A este último objeto accedemos mediante el método `edit()` de la clase `SharedPreferences`. Una vez obtenida la referencia al editor, utilizaremos los métodos `put` correspondientes al tipo de datos de cada preferencia para actualizar/insertar su valor, por ejemplo `putString(clave, valor)`, para actualizar una preferencia de tipo `String`. De forma análoga a los métodos `get` que ya hemos visto, tendremos disponibles métodos `put` para todos los tipos de datos básicos: `putInt()`, `putFloat()`, `putBoolean()`, etc. Finalmente, una vez actualizados/insertados todos los datos necesarios llamaremos al método `commit()` para confirmar los cambios. Veamos un ejemplo sencillo:

```
SharedPreferences prefs =
     getSharedPreferences("MisPreferencias",Context.MODE_PRIVATE);

SharedPreferences.Editor editor = prefs.edit();
editor.putString("email", "modificado@email.com");
editor.putString("nombre", "Prueba");
editor.commit();
```

¿Pero dónde se almacenan estas preferencias compartidas? Como dijimos al comienzo del apartado, las preferencias no se almacenan en ficheros binarios como las bases de datos SQLite, sino en ficheros XML. Estos ficheros XML se almacenan en una ruta que sigue el siguiente patrón:

`/data/data/paquetejava/shared_prefs/nombre_coleccion.xml`

Así, por ejemplo, en nuestro caso encontraríamos nuestro fichero de preferencias en la ruta:

`/data/data/net.sgoliver.android.preferences1/shared_prefs/MisPreferencias.xml`

Sirva una imagen del explorador de archivos del DDMS como prueba:

net.sgoliver.android.preferences1		2013-02-08	03:23	drwxr-x--x
cache		2013-02-08	03:23	drwxrwx--x
lib		2013-02-08	03:23	lrwxrwxrwx
shared_prefs		2013-02-08	03:23	drwxrwx--x
MisPreferencias.xml	159	2013-02-08	03:23	-rw-rw----

Si descargamos este fichero desde el DDMS y lo abrimos con cualquier editor de texto veremos un contenido como el siguiente:

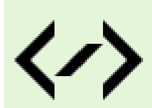
```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="nombre">prueba</string>
    <string name="email">modificado@email.com</string>
</map>
```

En este XML podemos observar cómo se han almacenado las dos preferencias de ejemplo que insertamos anteriormente, con sus claves y valores correspondientes.

Y nada más, así de fácil y práctico. Con esto hemos aprendido una forma sencilla de almacenar determinadas opciones de nuestra aplicación sin tener que recurrir para ello a definir bases de datos SQLite, que aunque tampoco añaden mucha dificultad sí que requieren algo más de trabajo por nuestra parte.

Se aporta una pequeña aplicación de ejemplo para este apartado que tan sólo incluye dos botones, el primero de ellos para guardar las preferencias tal como hemos descrito, y el segundo para recuperarlas y mostrarlas en el log.

En una segunda parte de este tema dedicado a las preferencias veremos cómo Android nos ofrece otra forma de gestionar estos datos, que se integra además fácilmente con la interfaz gráfica necesaria para solicitar los datos al usuario.



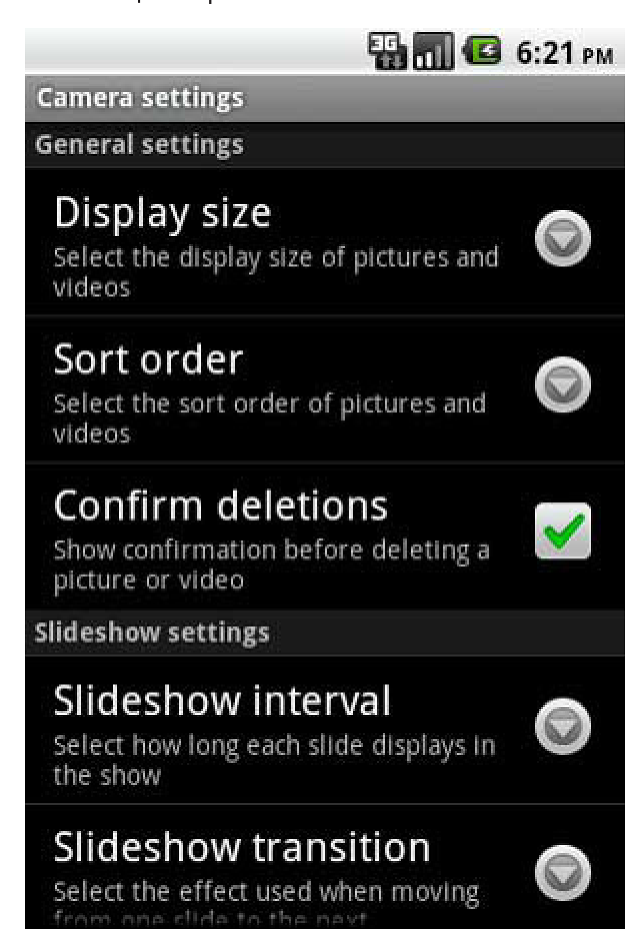
Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

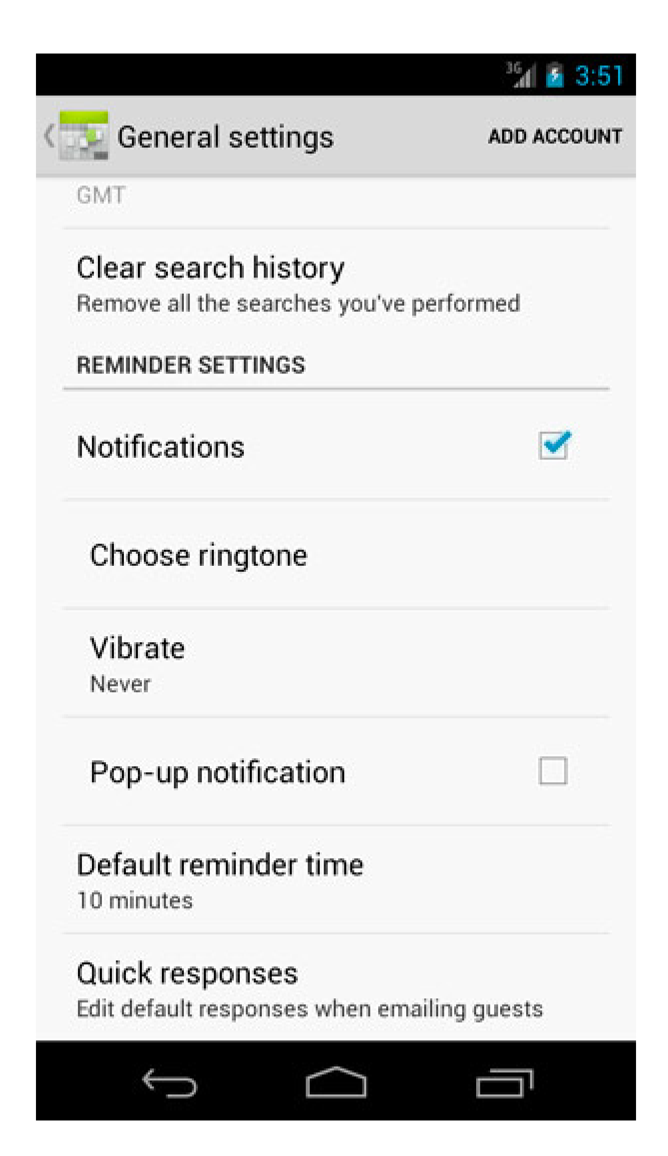
curso-android-src/android-preferences-1

Pantallas de Preferencias

En el apartado anterior hemos hablado de las *Shared Preferences*, un mecanismo que nos permite gestionar fácilmente las opciones de una aplicación permitiéndonos guardarlas en XML de una forma transparente para el programador. Y vimos cómo hacer uso de ellas mediante código, es decir, creando nosotros mismos los objetos necesarios (`SharedPreferences`) y añadiendo, modificando y/o recuperando "a mano" los valores de las opciones a través de los métodos correspondientes (`getString()`, `putString()`, …). Sin embargo, ya avisamos de que Android ofrece una forma alternativa de definir mediante XML un conjunto de opciones para una aplicación y crear por nosotros las pantallas necesarias para permitir al usuario modificarlas a su antojo. A esto dedicaremos este segundo apartado sobre preferencias.

Si nos fijamos en cualquier pantalla de preferencias estándar de Android veremos que todas comparten una interfaz común, similar por ejemplo a la que se muestra en las imágenes siguientes para Android 2.x y Android 4.x respectivamente:





Si atendemos por ejemplo a la primera imagen vemos cómo las diferentes opciones se organizan dentro de la **pantalla de opciones** en varias **categorías** ("*General Settings*" y "*Slideshow Settings*"). Dentro de cada categoría pueden aparecer varias opciones de diversos **tipos**, como por ejemplo de tipo checkbox ("*Confirm deletions*") o de tipo lista de selección ("*Display size*"). He resaltado las palabras "*pantalla de opciones*", "*categorías*", y "tipos de opción" porque serán estos los tres elementos principales con los que vamos a definir el conjunto de opciones o preferencias de nuestra aplicación. Empecemos.

Como hemos indicado, nuestra pantalla de opciones la vamos a definir mediante un XML, de forma similar a como definimos cualquier layout, aunque en este caso deberemos colocarlo en la carpeta `/res/xml`. El contenedor principal de nuestra pantalla de preferencias será el elemento `<PreferenceScreen>`. Este elemento representará a la pantalla de opciones en sí, dentro de la cual incluiremos el resto de elementos. Dentro de éste podremos incluir nuestra lista de opciones organizadas por categorías, que se representarán mediante el elemento `<PreferenceCategory>` al que daremos un texto descriptivo utilizando su atributo `android:title`. Dentro de cada categoría podremos añadir cualquier número de opciones, las cuales pueden ser de distintos tipos, entre los que destacan:

Tipo	Descripción
CheckBoxPreference	Marca seleccionable.
EditTextPreference	Cadena simple de texto.
ListPreference	Lista de valores seleccionables (exclusiva).
MultiSelectListPreference	Lista de valores seleccionables (múltiple).

Cada uno de estos tipos de preferencia requiere la definición de diferentes atributos, que iremos viendo en

los siguientes apartados.

CheckBoxPreference

Representa un tipo de opción que sólo puede tomar dos valores distintos: activada o desactivada. Es el equivalente a un control de tipo *checkbox*. En este caso tan sólo tendremos que especificar los atributos: nombre interno de la opción (`android:key`), texto a mostrar (`android:title`) y descripción de la opción (`android:summary`). Veamos un ejemplo:

```
<CheckBoxPreference
    android:key="opcion1"
    android:title="Preferencia 1"
    android:summary="Descripción de la preferencia 1" />
```

EditTextPreference

Representa un tipo de opción que puede contener como valor una cadena de texto. Al pulsar sobre una opción de este tipo se mostrará un cuadro de diálogo sencillo que solicitará al usuario el texto a almacenar. Para este tipo, además de los tres atributos comunes a todas las opciones (`key`, `title` y `summary`) también tendremos que indicar el texto a mostrar en el cuadro de diálogo, mediante el atributo `android:dialogTitle`. Un ejemplo sería el siguiente:

```
<EditTextPreference
   android:key="opcion2"
   android:title="Preferencia 2"
   android:summary="Descripción de la preferencia 2"
   android:dialogTitle="Introduce valor" />
```

ListPreference

Representa un tipo de opción que puede tomar como valor un elemento, y sólo uno, seleccionado por el usuario entre una lista de valores predefinida. Al pulsar sobre una opción de este tipo se mostrará la lista de valores posibles y el usuario podrá seleccionar uno de ellos. Y en este caso seguimos añadiendo atributos. Además de los cuatro ya comentados (`key`, `title`, `summary` y `dialogTitle`) tendremos que añadir dos más, uno de ellos indicando la lista de valores a visualizar en la lista y el otro indicando los valores internos que utilizaremos para cada uno de los valores de la lista anterior (Ejemplo: al usuario podemos mostrar una lista con los valores "*Español*" y "*Francés*", pero internamente almacenarlos como "*ESP*" y "*FRA*").

Estas listas de valores las definiremos también como ficheros XML dentro de la carpeta `/res/xml`. Definiremos para ello los recursos de tipos `<string-array>` necesarios, en este caso dos, uno para la lista de valores visibles y otro para la lista de valores internos, cada uno de ellos con su ID único correspondiente. Veamos cómo quedarían dos listas de ejemplo, en un fichero llamado "`codigospaises.xml`":

```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
   <string-array name="pais">
      <item>España</item>
      <item>Francia</item>
      <item>Alemania</item>
   </string-array>
   <string-array name="codigopais">
      <item>ESP</item>
      <item>FRA</item>
      <item>ALE</item>
   </string-array>
</resources>
```

En la preferencia utilizaremos los atributos `android:entries` y `android:entryValues` para hacer referencia a estas listas, como vemos en el ejemplo siguiente:

```
<ListPreference
   android:key="opcion3"
   android:title="Preferencia 3"
   android:summary="Descripción de la preferencia 3"
   android:dialogTitle="Indicar Pais"
   android:entries="@array/pais"
   android:entryValues="@array/codigopais" />
```

MultiSelectListPreference

[A partir de Android 3.0.x / Honeycomb] Las opciones de este tipo son muy similares a las `ListPreference`, con la diferencia de que el usuario puede seleccionar varias de las opciones de la lista de posibles valores. Los atributos a asignar son por tanto los mismos que para el tipo anterior.

```
<MultiSelectListPreference
   android:key="opcion4"
   android:title="Preferencia 4"
   android:summary="Descripción de la preferencia 4"
   android:dialogTitle="Indicar Pais"
   android:entries="@array/pais"
   android:entryValues="@array/codigopais" />
```

Como ejemplo completo, veamos cómo quedaría definida una pantalla de opciones con las 3 primeras opciones comentadas (ya que probaré con Android 2.2), divididas en 2 categorías llamadas por simplicidad "*Categoría 1*" y "*Categoría 2*". Llamaremos al fichero "`opciones.xml`".

```
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory android:title="Categoría 1">
        <CheckBoxPreference
            android:key="opcion1"
            android:title="Preferencia 1"
            android:summary="Descripción de la preferencia 1" />
        <EditTextPreference
            android:key="opcion2"
            android:title="Preferencia 2"
            android:summary="Descripción de la preferencia 2"
            android:dialogTitle="Introduce valor" />
    </PreferenceCategory>
    <PreferenceCategory android:title="Categoría 2">
        <ListPreference
            android:key="opcion3"
            android:title="Preferencia 3"
            android:summary="Descripción de la preferencia 3"
            android:dialogTitle="Indicar Pais"
            android:entries="@array/pais"
            android:entryValues="@array/codigopais" />
    </PreferenceCategory>
</PreferenceScreen>
```

Ya tenemos definida la estructura de nuestra pantalla de opciones, pero aún nos queda un paso más para poder hacer uso de ella desde nuestra aplicación. Además de la definición XML de la lista de opciones, debemos implementar una nueva actividad, que será a la que hagamos referencia cuando queramos mostrar nuestra pantalla de opciones y la que se encargará internamente de gestionar todas las opciones,

guardarlas, modificarlas, etc, a partir de nuestra definición XML.

Android nos facilita las cosas ofreciéndonos una clase de la que podemos derivar fácilmente la nuestra propia y que hace casi todo el trabajo por nosotros. Esta clase se llama `PreferenceActivity`. Tan sólo deberemos crear una nueva actividad (yo la he llamado `OpcionesActivity`) que extienda a esta clase, e implementar su evento `onCreate()` para añadir una llamada al método `addPreferencesFromResource()`, mediante el que indicaremos el fichero XML en el que hemos definido la pantalla de opciones. Lo vemos mejor directamente en el código:

```
public class OpcionesActivity extends PreferenceActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        addPreferencesFromResource(R.xml.opciones);
    }
}
```

Así de sencillo, nuestra nueva actividad, al extender a `PreferenceActivity`, se encargará por nosotros de crear la interfaz gráfica de nuestra lista de opciones según la hemos definido en el XML y se preocupará por nosotros de mostrar, modificar y guardar las opciones cuando sea necesario tras la acción del usuario.

Aunque esto continúa funcionando sin problemas en versiones recientes de Android, la API 11 trajo consigo una nueva forma de definir las pantallas de preferencias haciendo uso de fragments. Para ello, basta simplemente con definir la clase java del fragment, que deberá extender de `PreferenceFragment` y añadir a su método `onCreate()` una llamada a `addPreferencesFromResource()` igual que ya hemos visto antes.

```
public static class OpcionesFragment extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        addPreferencesFromResource(R.xml.opciones);
    }
}
```

Hecho esto ya no será necesario que la clase de nuestra pantalla de preferencias extienda de `PreferenceActivity`, sino que podrá ser una actividad normal. Para mostrar el fragment creado como contenido principal de la actividad utilizaríamos el *fragment manager* para sustituir el contenido de la pantalla (`android.R.id.content`) por el de nuestro fragment de preferencias recién definido:

```
public class SettingsActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        getFragmentManager().beginTransaction()
                .replace(android.R.id.content, new OpcionesFragment())
                .commit();
    }
}
```

Sea cual se la opción elegida para definir la pantalla de preferencias, el siguiente paso será añadir esta actividad al fichero `AndroidManifest.xml`, al igual que cualquier otra actividad que utilicemos en la aplicación.

```
<activity android:name=".OpcionesActivity"
    android:label="@string/app_name">
</activity>
```

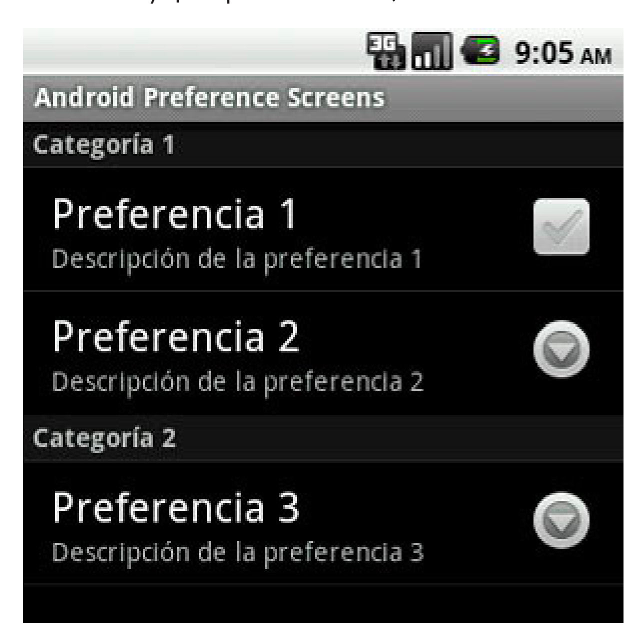
Ya sólo nos queda añadir a nuestra aplicación algún mecanismo para mostrar la pantalla de preferencias. Esta opción suele estar en un menú (para Android 2.x) o en el menú de overflow de la action bar (para Android 3 o superior), pero por simplificar el ejemplo vamos a añadir simplemente un botón (`btnPreferencias`) que abra la ventana de preferencias.

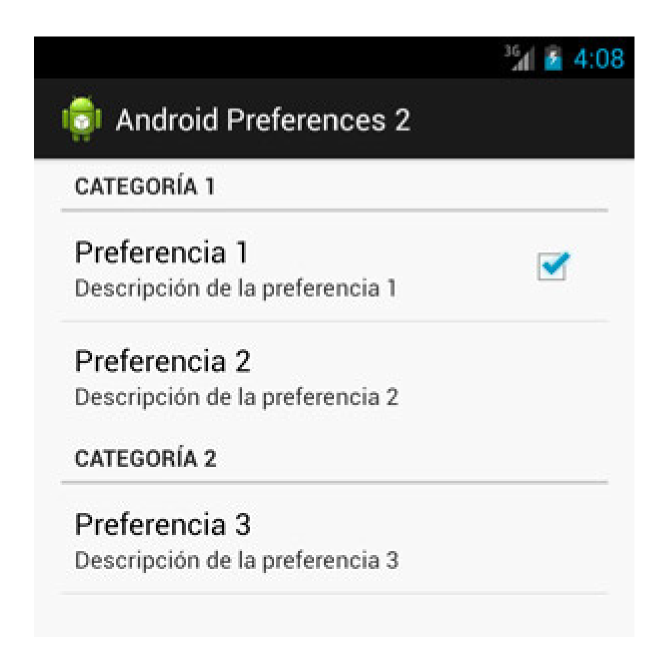
Al pulsar este botón llamaremos a la ventana de preferencias mediante el método `startActivity()`, como ya hemos visto en alguna ocasión, al que pasaremos como parámetros el contexto de la aplicación (nos vale con nuestra actividad principal) y la clase de la ventana de preferencias (`OpcionesActivity.class`).

```
btnPreferencias = (Button)findViewById(R.id.BtnPreferencias);

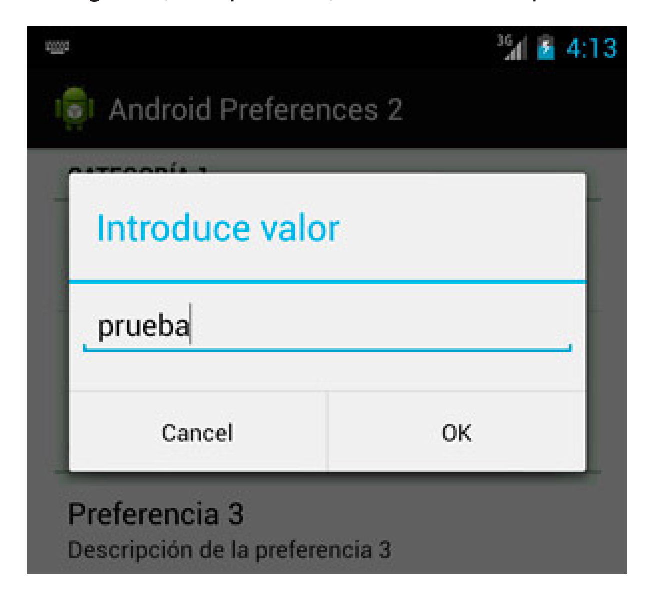
btnPreferencias.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        startActivity(new Intent(MainActivity.this,
                                 OpcionesActivity.class));
    }
});
```

Y esto es todo, ya sólo nos queda ejecutar la aplicación en el emulador y pulsar el botón de preferencias para mostrar nuestra nueva pantalla de opciones. Debe quedar como muestran las imágenes siguientes (para Android 2 y 4 respectivamente):

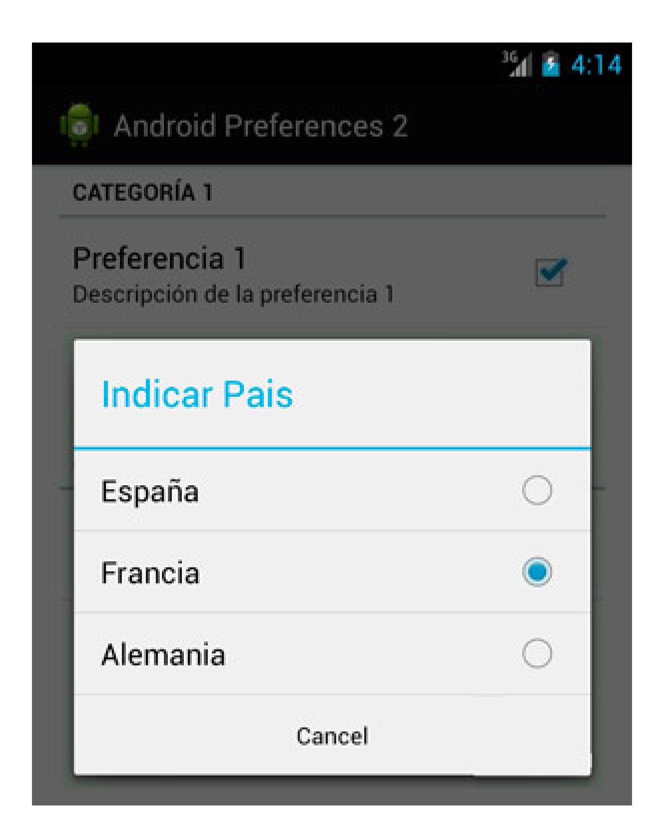




La primera opción podemos marcarla o desmarcarla directamente pulsando sobre la check de su derecha. La segunda, de tipo texto, nos mostrará al pulsarla un pequeño formulario para solicitar el valor de la opción.



Por último, la opción 3 de tipo lista, nos mostrará una ventana emergente con la lista de valores posibles, donde podremos seleccionar sólo uno de ellos.



Una vez establecidos los valores de las preferencias podemos salir de la ventana de opciones simplemente pulsando el botón Atrás del dispositivo o del emulador. Nuestra actividad `OpcionesActivity` se habrá ocupado por nosotros de guardar correctamente los valores de las opciones haciendo uso de la API de preferencias compartidas (*Shared Preferences*). Y para comprobarlo vamos a añadir otro botón (`btnObtenerOpciones`) a la aplicación de ejemplo que recupere el valor actual de las 3 preferencias y los escriba en el log de la aplicación.

La forma de acceder a las preferencias compartidas de la aplicación ya la vimos en el apartado anterior sobre este tema. Obtenemos la lista de preferencias mediante el método `getDefaultSharedPreferences()` y posteriormente utilizamos los distintos métodos `get()` para recuperar el valor de cada opción dependiendo de su tipo.

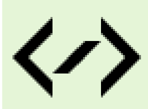
```
btnObtenerPreferencias.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        SharedPreferences pref =
            PreferenceManager.getDefaultSharedPreferences(
                AndroidPrefScreensActivity.this);

        Log.i("", "Opción 1: " + pref.getBoolean("opcion1", false));
        Log.i("", "Opción 2: " + pref.getString("opcion2", ""));
        Log.i("", "Opción 3: " + pref.getString("opcion3", ""));
    }
});
```

Si ejecutamos ahora la aplicación, establecemos las preferencias y pulsamos el nuevo botón de consulta que hemos creado veremos cómo en el log de la aplicación aparecen los valores correctos de cada preferencia. Se mostraría algo como lo siguiente:

```
10-08 09:27:09.681: INFO/(1162): Opción 1: true
10-08 09:27:09.681: INFO/(1162): Opción 2: prueba
10-08 09:27:09.693: INFO/(1162): Opción 3: FRA
```

Y hasta aquí hemos llegado con el tema de las preferencias, un tema muy interesante de controlar ya que casi ninguna aplicación se libra de hacer uso de ellas.Existen otras muchas opciones de configuración de las pantallas de preferencias, sobre todo con la llegada de Android 4, pero con lo que hemos visto aquí podremos cubrir la gran mayoría de casos.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-preferences-2

8

Localización Geográfica

VIII. Localización Geográfica

Localización Geográfica Básica

La localización geográfica en Android es uno de esos servicios que, a pesar de requerir poco código para ponerlos en marcha, no son para nada intuitivos ni fáciles de llegar a comprender por completo. Y esto no es debido al diseño de la plataforma Android en sí, sino a la propia naturaleza de este tipo de servicios. Por un lado, existen multitud de formas de obtener la localización de un dispositivo móvil, aunque la más conocida y popular es la localización por GPS, también es posible obtener la posición de un dispositivo por ejemplo a través de las antenas de telefonía móvil o mediante puntos de acceso Wi-Fi cercanos, y todos cada uno de estos mecanismos tiene una precisión, velocidad y consumo de recursos distinto. Por otro lado, el modo de funcionamiento de cada uno de estos mecanismos hace que su utilización desde nuestro código no sea todo lo directa e intuitiva que se desearía. Iremos comentando todo esto a lo largo del apartado, pero vayamos paso a paso.

¿Qué mecanismos de localización tenemos disponibles?

Lo primero que debe conocer una aplicación que necesite obtener la localización geográfica es qué mecanismos de localización (proveedores de localización, o *location providers*) tiene disponibles en el dispositivo. Como ya hemos comentado, los más comunes serán el GPS y la localización mediante la red de telefonía, pero podrían existir otros según el tipo de dispositivo.

La forma más sencilla de saber los proveedores disponibles en el dispositivo es mediante una llamada al método `getAllProviders()` de la clase `LocationManager`, clase principal en la que nos basaremos siempre a la hora de utilizar la API de localización de Android. Para ello, obtendremos una referencia al *location manager* llamando a `getSystemService(LOCATION_SERVICE)`, y posteriormente obtendremos la lista de proveedores mediante el método citado para obtener la lista de nombres de los proveedores:

```
LocationManager locManager =
    (LocationManager)getSystemService(LOCATION_SERVICE);
List<String> listaProviders = locManager.getAllProviders();
```

Una vez obtenida la lista completa de proveedores disponibles podríamos acceder a las propiedades de cualquiera de ellos (precisión, coste, consumo de recursos, o si es capaz de obtener la altitud, la velocidad, ...). Así, podemos obtener una referencia al *provider* mediante su nombre llamando al método `getProvider(nombre)` y posteriormente utilizar los métodos disponibles para conocer sus propiedades, por ejemplo `getAccuracy()` para saber su precisión (tenemos disponibles las constantes `Criteria.ACCURACY_FINE` para precisión alta, y `Criteria.ACCURACY_COARSE` para precisión media), `supportsAltitude()` para saber si obtiene la altitud, o `getPowerRequirement()` para obtener el nivel de consumo de recursos del proveedor. La lista completa de métodos para obtener las características de un proveedor se puede consultar en la documentación oficial de la clase `LocationProvider`.

```
LocationManager locManager =
    (LocationManager)getSystemService(LOCATION_SERVICE);
List<String> listaProviders = locManager.getAllProviders();

LocationProvider provider = locManager.getProvider(listaProviders.get(0));
int precision = provider.getAccuracy();
boolean obtieneAltitud = provider.supportsAltitude();
int consumoRecursos = provider.getPowerRequirement();
```

Al margen de esto, hay que tener en cuenta que la lista de proveedores devuelta por el método `getAllProviders()` contendrá todos los proveedores de localización conocidos por el dispositivo, incluso si éstos no están permitidos (según los permisos de la aplicación) o no están activados, por lo que

esta información puede que no nos sea de mucha ayuda.

¿Qué proveedor de localización es mejor para mi aplicación?

Android proporciona un mecanismo alternativo para obtener los proveedores que cumplen unos determinados requisitos entre todos los disponibles. Para ello nos permite definir un criterio de búsqueda, mediante un objeto de tipo `Criteria`, en el que podremos indicar las características mínimas del proveedor que necesitamos utilizar (podéis consultar la documentación oficial de la clase `Criteria` para saber todas las características que podemos definir). Así, por ejemplo, para buscar uno con precisión alta y que nos proporcione la altitud definiríamos el siguiente criterio de búsqueda:

```
Criteria req = new Criteria();
req.setAccuracy(Criteria.ACCURACY_FINE);
req.setAltitudeRequired(true);
```

Tras esto, podremos utilizar los métodos `getProviders()` o `getBestProvider()` para obtener la lista de proveedores que se ajustan mejor al criterio definido o el proveedor que mejor se ajusta a dicho criterio, respectivamente. Además, ambos métodos reciben un segundo parámetro que indica si queremos que sólo nos devuelvan proveedores que están activados actualmente. Veamos cómo se utilizarían estos métodos:

```
//Mejor proveedor por criterio
String mejorProviderCrit = locManager.getBestProvider(req, false);

//Lista de proveedores por criterio
List<String> listaProvidersCrit = locManager.getProviders(req, false);
```

Con esto, ya tenemos una forma de seleccionar en cada dispositivo aquel proveedor que mejor se ajusta a nuestras necesidades.

¿Está disponible y activado un proveedor determinado?

Aunque, como ya hemos visto, tenemos la posibilidad de buscar dinámicamente proveedores de localización según un determinado criterio de búsqueda, es bastante común que nuestra aplicación esté diseñada para utilizar uno en concreto, por ejemplo el GPS, y por tanto necesitaremos algún mecanismo para saber si éste está activado o no en el dispositivo. Para esta tarea, la clase `LocationManager` nos proporciona otro método llamado `isProviderEnabled()` que nos permite hacer exactamente lo que necesitamos. Para ello, debemos pasarle el nombre del *provider* que queremos consultar. Para los más comunes tenemos varias constantes ya definidas:

- `LocationManager.NETWORK_PROVIDER`. Localización por la red de telefonía.
- `LocationManager.GPS_PROVIDER`. Localización por GPS.

De esta forma, si quisiéramos saber si el GPS está habilitado o no en el dispositivo (y actuar en consecuencia), haríamos algo parecido a lo siguiente:

```
//Si el GPS no está habilitado
if (!locManager.isProviderEnabled(LocationManager.GPS_PROVIDER)) {
     mostrarAvisoGpsDeshabilitado();
}
```

En el código anterior, verificamos si el GPS está activado y en caso negativo mostramos al usuario un mensaje de advertencia. Este mensaje podríamos mostrarlo sencillamente en forma de notificación de tipo *toast*, pero en el próximo apartado sobre localización veremos cómo podemos, además de informar de que el GPS está desactivado, invitar al usuario a activarlo dirigiéndolo automáticamente a la pantalla de configuración del dispositivo.

El GPS ya está activado, ¿y ahora qué?

Una vez que sabemos que nuestro proveedor de localización favorito está activado, ya estamos en disposición de intentar obtener nuestra localización actual. Y aquí es donde las cosas empiezan a ser menos intuitivas. Para empezar, en Android no existe ningún método del tipo "obtenerPosiciónActual()". Obtener la posición a través de un dispositivo de localización como por ejemplo el GPS no es una tarea inmediata, sino que puede requerir de un cierto tiempo de procesamiento y de espera, por lo que no tendría sentido proporcionar un método de ese tipo.

Si buscamos entre los métodos disponibles en la clase `LocationManager`, lo más parecido que encontramos es un método llamado `getLastKnownLocation(String provider)`, que como se puede suponer por su nombre, nos devuelve la última posición conocida del dispositivo devuelta por un provider determinado. Es importante entender esto: este método NO devuelve la posición actual, este método NO solicita una nueva posición al proveedor de localización, este método se limita a devolver la última posición que se obtuvo a través del proveedor que se le indique como parámetro. Y esta posición se pudo obtener hace pocos segundos, hace días, hace meses, o incluso nunca (si el dispositivo ha estado apagado, si nunca se ha activado el GPS, ...). Por tanto, cuidado cuando se haga uso de la posición devuelta por el método `getLastKnownLocation()`.

Entonces, ¿de qué forma podemos obtener la posición real actualizada? Pues la forma correcta de proceder va a consistir en algo así como activar el proveedor de localización y suscribirnos a sus notificaciones de cambio de posición. O dicho de otra forma, vamos a suscribirnos al evento que se lanza cada vez que un proveedor recibe nuevos datos sobre la localización actual. Y para ello, vamos a darle previamente unas indicaciones (que no ordenes, ya veremos esto en el próximo apartado) sobre cada cuanto tiempo o cada cuanta distacia recorrida necesitaríamos tener una actualización de la posición.

Todo esto lo vamos a realizar mediante una llamada al método `requestLocationUpdates()`, al que deberemos pasar 4 parámetros distintos:

- Nombre del proveedor de localización al que nos queremos suscribir.
- Tiempo mínimo entre actualizaciones, en milisegundos.
- Distancia mínima entre actualizaciones, en metros.
- Instancia de un objeto `LocationListener`, que tendremos que implementar previamente para definir las acciones a realizar al recibir cada nueva actualización de la posición.

Tanto el tiempo como la distancia entre actualizaciones pueden pasarse con valor 0, lo que indicaría que ese criterio no se tendrá en cuenta a la hora de decidir la frecuencia de actualizaciones. Si ambos valores van a cero, las actualizaciones de posición se recibirán tan pronto y tan frecuentemente como estén disponibles. Además, como ya hemos indicado, es importante comprender que tanto el tiempo como la distancia especificadas se entenderán como simples indicaciones o "pistas" para el proveedor (al menos para versiones no recientes de Android), por lo que puede que no se cumplan de forma estricta. En el próximo apartado intentaremos ver esto con más detalle para entenderlo mejor. Por ahora nos basta con esta información.

En cuanto al *listener*, éste será del tipo `LocationListener` y contendrá una serie de métodos asociados a los distintos eventos que podemos recibir del proveedor:

- `onLocationChanged(location)`. Lanzado cada vez que se recibe una actualización de la posición.
- `onProviderDisabled(provider)`. Lanzado cuando el proveedor se deshabilita.
- `onProviderEnabled(provider)`. Lanzado cuando el proveedor se habilita.
- `onStatusChanged(provider, status, extras)`. Lanzado cada vez que el proveedor cambia su estado, que puede variar entre `OUT_OF_SERVICE`, `TEMPORARILY_UNAVAILABLE`, `AVAILABLE`.

Por nuestra parte, tendremos que implementar cada uno de estos métodos para responder a los eventos del proveedor, sobre todo al más interesante, `onLocationChanged()`, que se ejecutará cada vez que se recibe una nueva localización desde el proveedor. Veamos un ejemplo de cómo implementar un listener de este tipo:

```
LocationListener locListener = new LocationListener() {

    public void onLocationChanged(Location location) {
        mostrarPosicion(location);
    }

    public void onProviderDisabled(String provider){
        lblEstado.setText("Provider OFF");
    }

    public void onProviderEnabled(String provider){
        lblEstado.setText("Provider ON");
    }

    public void onStatusChanged(String provider, int status, Bundle extras){
        lblEstado.setText("Provider Status: " + status);
    }
};
```

Como podéis ver, en nuestro caso de ejemplo nos limitamos a mostrar al usuario la información recibida en el evento, bien sea un simple cambio de estado, o una nueva posición, en cuyo caso llamamos al método auxiliar `mostrarPosicion()` para refrescar todos los datos de la posición en la pantalla. Para este ejemplo hemos construido una interfaz muy sencilla, donde se muestran 3 datos de la posición (latitud, longitud y precisión) y un campo para mostrar el estado del proveedor. Además, se incluyen dos botones para comenzar y detener la recepción de nuevas actualizaciones de la posición. No incluyo aquí el código de la interfaz para no alargar más el apartado, pero puede consultarse en el código fuente suministrado al final del texto. El aspecto de nuestra ventana es el siguiente:



En el método `mostrarPosicion()` nos vamos a limitar a mostrar los distintos datos de la posición pasada como parámetro en los controles correspondientes de la interfaz, utilizando para ello los métodos proporcionados por la clase `Location`. En nuestro caso utilizaremos `getLatitude()`, `getAltitude()` y `getAccuracy()` para obtener la latitud, longitud y precisión respectivamente. Por supuesto, hay otros métodos disponibles en esta clase para obtener la altura, orientación, velocidad, etc… que se pueden consultar en la documentación oficial de la clase `Location`. Veamos el código:

```
private void mostrarPosicion(Location loc) {
    if(loc != null)
    {
        lblLatitud.setText("Latitud: " + String.valueOf(loc.getLatitude()));
        lblLongitud.setText("Longitud: " +
                    String.valueOf(loc.getLongitude()));
        lblPrecision.setText("Precision: " +
                    String.valueOf(loc.getAccuracy()));
    }
    else
    {
        lblLatitud.setText("Latitud: (sin_datos)");
        lblLongitud.setText("Longitud: (sin_datos)");
        lblPrecision.setText("Precision: (sin_datos)");
    }
}
```

¿Por qué comprobamos si la localización recibida es `null`? Como ya hemos dicho anteriormente, no tenemos mucho control sobre el momento ni la frecuencia con la que vamos a recibir las actualizaciones de posición desde un proveedor, por lo que tampoco estamos seguros de tenerlas disponibles desde un primer momento. Por este motivo, una técnica bastante común es utilizar la posición que devuelve el método `getLastKnownLocation()` como posición "provisional" de partida y a partir de ahí esperar a recibir la primera actualización a través del `LocationListener`. Y como también dijimos, la última posición conocida podría no existir en el dispositivo, de ahí que comprobemos si el valor recibido es `null`. Para entender mejor esto, a continuación tenéis la estructura completa del método que lanzamos al comenzar la recepción de actualizaciones de posición (al pulsar el botón "*Activar*" de la interfaz):

```
private void comenzarLocalizacion()
{
    //Obtenemos una referencia al LocationManager
    locManager =
        (LocationManager)getSystemService(Context.LOCATION_SERVICE);

    //Obtenemos la última posición conocida
    Location loc =
        locManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);

    //Mostramos la última posición conocida
    mostrarPosicion(loc);

    //Nos registramos para recibir actualizaciones de la posición
    locListener = new LocationListener() {
        public void onLocationChanged(Location location) {
            mostrarPosicion(location);
        }

        //Resto de métodos del listener
        //...
    };

    locManager.requestLocationUpdates(
        LocationManager.GPS_PROVIDER, 30000, 0, locListener);
}
```

Como se puede observar, al comenzar la recepción de posiciones, mostramos en primer lugar la última posición conocida, y posteriormente solicitamos al GPS actualizaciones de posición cada 30 segundos.

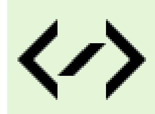
Por último, nos quedaría únicamente comentar cómo podemos detener la recepción de nuevas actualizaciones

de posición. Algo que es tan sencillo como llamar al método `removeUpdates()` del *location manager*. De esta forma, la implementación del botón "*Desactivar*" sería tan sencilla como esto:

```
btnDesactivar.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        locManager.removeUpdates(locListener);
    }
});
```

Con esto habríamos concluido nuestra aplicación de ejemplo. Sin embargo, si descargáis el código completo del apartado y ejecutáis la aplicación en el emulador veréis que, a pesar funcionar todo correctamente, sólo recibiréis una lectura de la posición (incluso puede que ninguna). Esto es debido a que la ejecución y prueba de aplicaciones de este tipo en el emulador de Android, al no tratarse de un dispositivo real y no estar disponible un receptor GPS, requiere de una serie de pasos adicionales para simular cambios en la posición del dispositivo.

Todo esto, además de algunas aclaraciones que nos han quedado pendientes en esta primera entrega sobre localización, lo veremos en el próximo apartado. Por ahora os dejo el código fuente completo para que podáis hacer vuestras propias pruebas.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-localizacion-1

Profundizando en la Localización Geográfica

En el apartado anterior comentamos los pasos básicos necesarios para construir aplicaciones que accedan a la posición geográfica del dispositivo. Ya comentamos algunas particularidades de este servicio de la API de Android, pero dejamos en el tintero algunas aclaraciones más detalladas y un tema importante y fundamental, como es la depuración de este tipo de aplicaciones que manejan datos de localización. En este nuevo apartado intentaré abarcar todos estos temas.

Como base para este apartado voy a utilizar la misma aplicación de ejemplo que construimos en la anterior entrega, haciendo tan sólo unas pequeñas modificaciones:

- Reduciremos el tiempo entre actualizaciones de posición a la mitad, 15 segundos, para evitar tiempos de espera demasiado largos durante la ejecución de la aplicación.
- Generaremos algunos mensajes de log en puntos clave del código para poder estudiar con más detalle el comportamiento de la aplicación en tiempo de ejecución.

La generación de mensajes de log resulta ser una herramienta perfecta a la hora de depurar aplicaciones del tipo que estamos tratando, ya que en estos casos el código no facilita demasiado la depuración típica paso a paso que podemos realizar en otras aplicaciones.

En nuestro caso de ejemplo sólo vamos a generar mensajes de log cuando ocurran dos circunstancias:

- Cuando el proveedor de localización cambie de estado, evento `onStatusChanged()`, mostraremos el nuevo estado.
- Cuando se reciba una nueva actualización de la posición, evento `onLocationChanged()`, mostraremos las nuevas coordenadas recibidas.

Nuestro código quedaría por tanto tal como sigue:

```
private void actualizarPosicion()
{
    //Obtenemos una referencia al LocationManager
    locationManager =
        (LocationManager)getSystemService(Context.LOCATION_SERVICE);

    //Obtenemos la última posición conocida
    Location location =
        locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);

    //Mostramos la última posición conocida
    muestraPosicion(location);


    //Nos registramos para recibir actualizaciones de la posición
    locationListener = new LocationListener() {
        public void onLocationChanged(Location location) {
            muestraPosicion(location);
        }
        public void onProviderDisabled(String provider){
            lblEstado.setText("Provider OFF");
        }
        public void onProviderEnabled(String provider){
            lblEstado.setText("Provider ON");
        }
        public void onStatusChanged(String provider, int status,
                                    Bundle extras){
            Log.i("LocAndroid", "Provider Status: " + status);
            lblEstado.setText("Provider Status: " + status);
        }
    };

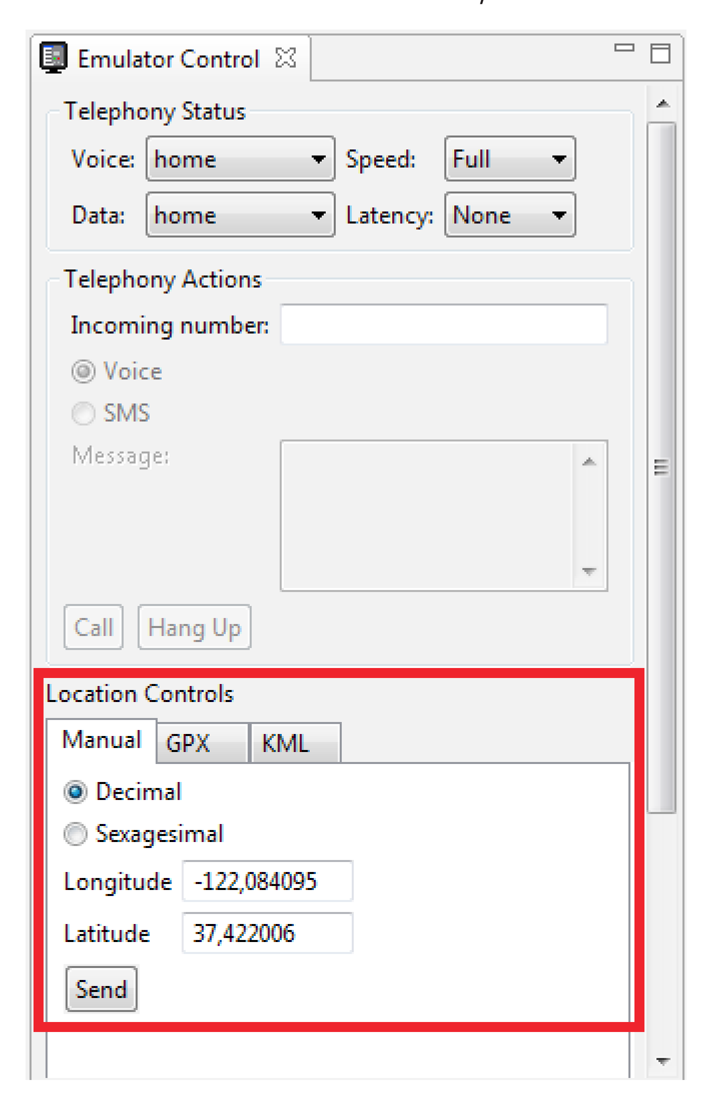
    locationManager.requestLocationUpdates(
            LocationManager.GPS_PROVIDER, 15000, 0, locationListener);
}

private void muestraPosicion(Location loc) {
    if(loc != null) {
        lblLatitud.setText("Latitud: " + String.valueOf(loc.getLatitude()));
        lblLongitud.setText("Longitud: " +
                            String.valueOf(loc.getLongitude()));
        lblPrecision.setText("Precision: " +
                             String.valueOf(loc.getAccuracy()));
        Log.i("LocAndroid", String.valueOf(
            loc.getLatitude() + " - " + String.valueOf(loc.getLongitude())));
    } else {
        lblLatitud.setText("Latitud: (sin_datos)");
        lblLongitud.setText("Longitud: (sin_datos)");
        lblPrecision.setText("Precision: (sin_datos)");
    }
}
```

Si ejecutamos en este momento la aplicación en el emulador y pulsamos el botón "*Activar*" veremos cómo los cuadros de texto se rellenan con la información de la última posición conocida (si existe), pero sin embargo estos datos no cambiarán en ningún momento ya que por el momento el emulador de Android tan sólo cuenta con esa información. ¿Cómo podemos simular la actualización de la posición del dispositivo para ver si nuestra aplicación responde exactamente como esperamos?

Pues bien, para hacer esto tenemos varias opciones. La primera de ellas, y la más sencilla, es el envío manual

de una nueva posición al emulador de Android, para simular que éste hubiera cambiado su localización. Esto se puede realizar desde la perspectiva de *DDMS*, en la pestaña *Emulator Control*, donde podemos encontrar una sección llamada *Location Controls*, mostrada en la imagen siguiente:



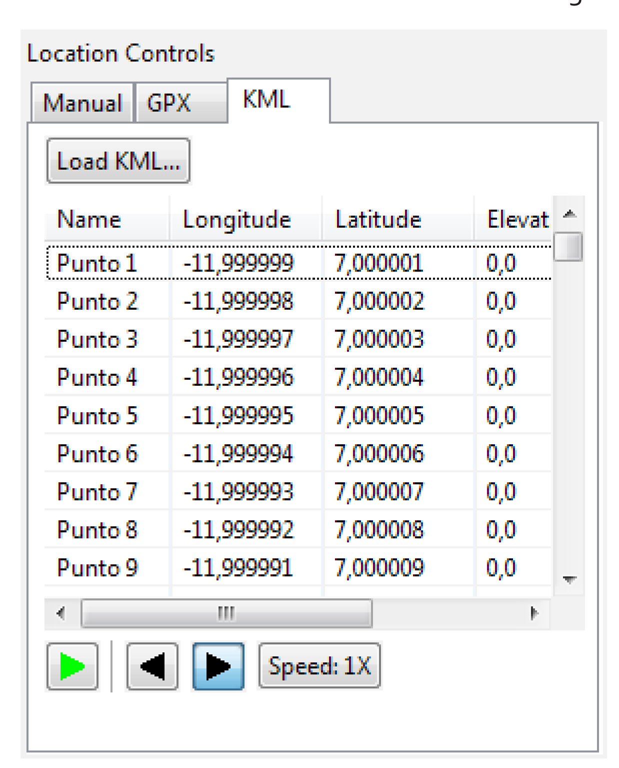
Con estos controles podemos enviar de forma manual al emulador en ejecución unas nuevas coordenadas de posición, para simular que éstas se hubieran recibido a través del proveedor de localización utilizado. De esta forma, si introducimos unas coordenadas de longitud y latitud y pulsamos el botón "*Send*" mientras nuestra aplicación se ejecuta en el emulador, esto provocará la ejecución del evento `onLocationChanged()` y por consiguiente se mostrarán estos mismos datos en sus controles correspondientes de la interfaz, como vemos en la siguiente captura de pantalla:



Por supuesto, si hacemos nuevos envíos de coordenadas desde Eclipse veremos cómo ésta se va actualizando en nuestra aplicación sin ningún tipo de problemas. Sin embargo este método de manual no resulta demasiado adecuado ni cómodo para probar toda la funcionalidad de nuestra aplicación, por ejemplo la actualización de posición cada 15 segundos.

Por ello, Android proporciona otro método algo menos manual de simular cambios frecuentes de posición para probar nuestras aplicaciones. Este método consiste en proporcionar, en vez de una sola coordenada cada vez, una lista de coordenadas que se irán enviando automáticamente al emulador una tras otra a una determinada velocidad, de forma que podamos simular que el dispositivo se mueve constantemente y que nuestra aplicación responde de forma correcta y en el momento adecuado a esos cambios de posición. Y esta lista de coordenadas se puede proporcionar de dos formas distintas, en formato GPX o como fichero KML. Ambos tipos de fichero son ampliamente utilizados por aplicaciones y dispositivos de localización, como GPS, aplicaciones de cartografía y mapas, etc. Los ficheros KML podemos generarlos por ejemplo a través de la aplicación *Google Earth* o manualmente con cualquier editor de texto, pero recomiendo consultar los dos enlaces anteriores para obtener más información sobre cada formato. Para este ejemplo, yo he generado un fichero KML de muestra con una lista de 1000 posiciones geográficas al azar.

Para utilizar este fichero como fuente de datos para simular cambios en la posición del dispositivo, accedemos nuevamente a los *Location Controls* y pulsamos sobre la pestaña GPX o KML, según el formato que hayamos elegido, que en nuestro caso será KML. Pulsamos el botón "*Load KML...*" para seleccionar nuestro fichero y veremos la lista de coordenadas como en la siguiente imagen:



Una vez cargado el fichero, tendremos disponibles los cuatro botones inferiores para (de izquierda a derecha):

- Avanzar automáticamente por la lista.
- Ir a la posición anterior de la lista de forma manual.
- Ir a la posición siguiente de la lista de forma manual.
- Establecer la velocidad de avance automático.

Entendido esto, vamos a utilizar la lista de posiciones para probar nuestra aplicación. Para ello, ejecutamos la aplicación en el emulador, pulsamos nuestro botón "*Activar*" para comenzar a detectar cambios de posición, y pulsamos el botón de avance automático (botón verde) que acabamos de comentar.

Llegados a este punto pueden ocurrir varias cosas, dependiendo del dispositivo o el emulador que estemos utilizando, y por supuesto de la versión de Android sobre la que ejecutemos el ejemplo. O bien todo funciona según lo esperado y empezamos a recibir una lectura de posición cada 15 segundos, o bien pueden aparecernos varias actualizaciones con una frecuencia mucho menor que el periodo especificado. La primera situación parece lógica, es lo que habíamos pedido. ¿Pero cómo es posible la segunda? ¿No habíamos configurado el `LocationListener` para obtener actualizaciones de posición cada 15 segundos? ¿Por qué llegan más actualizaciones de las esperadas? Antes de contestar a esto, dejemos por ejemplo que la aplicación se ejecute durante un minuto sobre un emulador con versión 2.2 de Android (API 8). Tras unos 60 segundos de ejecución detenemos la captura de posiciones pulsando nuestro botón "*Desactivar*".

Ahora vayamos a la ventana de log del DDMS y veamos los mensajes de log ha generado nuestra aplicación para intentar saber qué ha ocurrido. En mi caso, los mensajes generados son los siguientes (en tu caso deben ser muy parecidos):

```
05-08 10:50:37.921: INFO/LocAndroid(251): 7.0 - -11.999998333333334
05-08 10:50:38.041: INFO/LocAndroid(251): Provider Status: 2
05-08 10:50:38.901: INFO/LocAndroid(251): 7.000001666666666 - -11.999996666666668
05-08 10:50:39.941: INFO/LocAndroid(251): 7.000001666666666 - -11.999996666666668
05-08 10:50:41.011: INFO/LocAndroid(251): 7.000003333333333 - -11.999995000000002
05-08 10:50:43.011: INFO/LocAndroid(251): 7.000005000000001 - -11.999993333333334
05-08 10:50:45.001: INFO/LocAndroid(251): 7.000006666666667 - -11.999991666666665
05-08 10:50:46.061: INFO/LocAndroid(251): 7.000008333333333 - -11.999989999999999
05-08 10:50:47.131: INFO/LocAndroid(251): 7.000008333333333 - -11.999989999999999
05-08 10:50:47.182: INFO/LocAndroid(251): Provider Status: 1
05-08 10:51:02.232: INFO/LocAndroid(251): 7.000023333333333 - -11.999975
05-08 10:51:02.812: INFO/LocAndroid(251): 7.000023333333333 - -11.999973333333333
05-08 10:51:02.872: INFO/LocAndroid(251): Provider Status: 2
05-08 10:51:03.872: INFO/LocAndroid(251): 7.000024999999999 - -11.999973333333333
05-08 10:51:04.912: INFO/LocAndroid(251): 7.000026666666668 - -11.999971666666665
05-08 10:51:05.922: INFO/LocAndroid(251): 7.000026666666668 - -11.999971666666665
05-08 10:51:06.982: INFO/LocAndroid(251): 7.000028333333334 - -11.99997
05-08 10:51:08.032: INFO/LocAndroid(251): 7.000028333333334 - -11.999968333333333
05-08 10:51:09.062: INFO/LocAndroid(251): 7.00003 - -11.999968333333333
05-08 10:51:10.132: INFO/LocAndroid(251): 7.000031666666667 - -11.999966666666667
05-08 10:51:12.242: INFO/LocAndroid(251): 7.000033333333333 - -11.999965000000001
05-08 10:51:13.292: INFO/LocAndroid(251): 7.000033333333333 - -11.999963333333335
05-08 10:51:13.342: INFO/LocAndroid(251): Provider Status: 1
05-08 10:51:28.372: INFO/LocAndroid(251): 7.000048333333333 - -11.999950000000002
05-08 10:51:28.982: INFO/LocAndroid(251): 7.000048333333333 - -11.999950000000002
05-08 10:51:29.032: INFO/LocAndroid(251): Provider Status: 2

05-08 10:51:30.002: INFO/LocAndroid(251): 7.000050000000001 - -11.999948333333334
05-08 10:51:31.002: INFO/LocAndroid(251): 7.000051666666667 - -11.999946666666665
05-08 10:51:33.111: INFO/LocAndroid(251): 7.000053333333333 - -11.999944999999999
05-08 10:51:34.151: INFO/LocAndroid(251): 7.000053333333333 - -11.999944999999999
05-08 10:51:35.201: INFO/LocAndroid(251): 7.000055 - -11.999943333333333
05-08 10:51:36.251: INFO/LocAndroid(251): 7.0000566666666675 -
-11.999941666666667
05-08 10:51:37.311: INFO/LocAndroid(251): 7.0000566666666675 -
-11.999941666666667
05-08 10:51:38.361: INFO/LocAndroid(251): 7.0000583333333335 - -11.99994
05-08 10:51:38.431: INFO/LocAndroid(251): Provider Status: 1
```

Estudiemos un poco este log. Si observamos las marcas de fecha hora vemos varias cosas:

- Un primer grupo de actualizaciones entre las 10:50:37 y las 10:50:47, con 8 lecturas.
- Un segundo grupo de actualizaciones entre las 10:51:02 y las 10:51:13, con 11 lecturas.

- Un tercer grupo de actualizaciones entre las 10:51:28 y las 10:51:38, con 10 lecturas.
- Entre cada grupo de lecturas transcurren aproximadamente 15 segundos.
- Los grupos están formados por un número variable de lecturas.

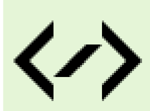
Por tanto ya podemos sacar algunas conclusiones. Indicar al *location listener* una frecuencia de 15 segundos entre actualizaciones no quiere decir que vayamos a tener una sola lectura cada 15 segundos, sino que al menos tendremos una nueva con dicha frecuencia. Sin embargo, como podemos comprobar en los *logs*, las lecturas se recibirán por grupos separados entre sí por el intervalo de tiempo indicado.

Más conclusiones, ahora sobre el estado del proveedor de localización. Si buscamos en el log los momentos donde cambia el estado del proveedor vemos dos cosas importantes:

- Después de recibir cada grupo de lecturas el proveedor pasa a estado 1 (`TEMPORARILY_UNAVAILABLE`).
- Tras empezar a recibir de nuevo lecturas el proveedor pasa a estado 2 (`AVAILABLE`).

Estos cambios en el estado de los proveedores de localización pueden ayudarnos a realizar diversas tareas. Un ejemplo típico es utilizar el cambio de estado a 1 (es decir, cuando se ha terminado de recibir un grupo de lecturas) para seleccionar la lectura más precisa del grupo recibido, algo especialmente importante cuando se están utilizando varios proveedores de localización simultáneamente, cada uno con una precisión distinta.

A modo de resumen, en este apartado hemos visto cómo podemos utilizar las distintas herramientas que proporciona la plataforma Android y el entorno de desarrollo Eclipse para simular cambios de posición del dispositivo durante la ejecución de nuestras aplicaciones en el emulador. También hemos visto cómo la generación de mensajes de log puede ayudarnos a depurar este tipo de aplicaciones, y finalmente hemos utilizado esta herramienta de depuración para entender mejor el funcionamiento de los *location listener* y el comportamiento de los proveedores de localización.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-localizacion-2

9

Mapas

IX. Mapas en Android

Preparativos y ejemplo básico

En la edición anterior de este libro hablaba sobre la API v1 de Google Maps para Android (capítulos que he mantenido como referencia en la versión web de este curso), pero a finales de 2012 Google presentó la segunda versión de su API de Google Maps para Android, y será ésta la que recoja en esta nueva edición del manual.

Esta nueva versión presenta muchas novedades interesantes, de las que cabe destacar las siguientes:

- Integración con los Servicios de Google Play (*Google Play Services*) y la Consola de APIs.
- Utilización a través de un nuevo tipo específico de fragment (`MapFragment`), una mejora muy esperada por muchos.
- Utilización de *mapas vectoriales*, lo que repercute en una mayor velocidad de carga y una mayor eficiencia en cuanto a uso de ancho de banda.
- Mejoras en el sistema de caché, lo que reducirá en gran medida las famosas áreas en blanco que tardan en cargar.
- Los mapas son ahora 3D, es decir, podremos mover nuestro punto de vista de forma que lo veamos en perspectiva.

Al margen de las novedades generales, como desarrolladores ¿qué diferencias nos vamos a encontrar con respecto a la API anterior a la hora de desarrollar nuestras aplicaciones Android?

Pues la principal será el componente que utilizaremos para la inclusión de mapas en nuestra aplicación. Si recordamos la anterior versión de la API, para incluir un mapa en la aplicación debíamos utilizar un control de tipo `MapView`, que además requería que su actividad contenedora fuera del tipo `MapActivity`. Con la nueva API nos olvidaremos de estos dos componentes y pasaremos a tener sólo uno, un nuevo tipo específico de fragment llamado `MapFragment`. Esto nos permitirá entre otras cosas añadir uno [o varios, esto también es una novedad] mapas a cualquier actividad, sea del tipo que sea, y contando por supuesto con todas las ventajas del uso de *fragments*. **Nota importante**: dado que el nuevo control de mapas se basa en fragments, si queremos mantener la compatibilidad con versiones de Android anteriores a la 3.0 tendremos que utilizar la librería de soporte `android-support`. Más adelante veremos más detalles sobre esto.

Además de esta novedad, la integración de la API con los *Google Play Services* y la *Consola de APIs de Google*, harán que los preparativos del entorno, las librerías utilizadas, y el proceso de obtención de la *API Key* de Google Maps sean un poco distintos a los que se utilizaban para la primera versión.

Pues bien, en este nuevo capítulo del curso vamos a describir los pasos necesarios para hacer uso de la nueva versión de la API de mapas de Google (Google Maps Android API v2).

Como en los artículos previos donde aprendimos a utilizar la API v1, en este caso también será necesario realizar algunos preparativos y tareas previas antes de poder empezar a utilizarlos en nuestras aplicaciones.

En primer lugar, dado que la API v2 se proporciona como parte del SDK de *Google Play Services*, será necesario incorporar previamente a nuestro entorno de desarrollo dicho paquete. Haremos esto accediendo desde Eclipse al Android SDK Manager y descargando del apartado de extras el paquete llamado "Google Play Services".

Android SDK Manager

Packages Tools

SDK Path: C:\Android\sdk-r21

Packages

Name	API	Rev.	Status
Android 1.5 (API 3)			
Extras			
Android Support Library		11	Installed
Google AdMob Ads SDK		8	Not installed
Google Analytics SDK		2	Not installed
Google Cloud Messaging for Android Library		3	Installed
Google Play services		2	Not installed
Google Play APK Expansion Library		2	Not installed
Google Play Billing Library		2	Not installed
Google Play Licensing Library		2	Not installed
Google USB Driver		7	Not installed
Google Web Driver		2	Not installed
Intel x86 Emulator Accelerator (HAXM)		2	Installed

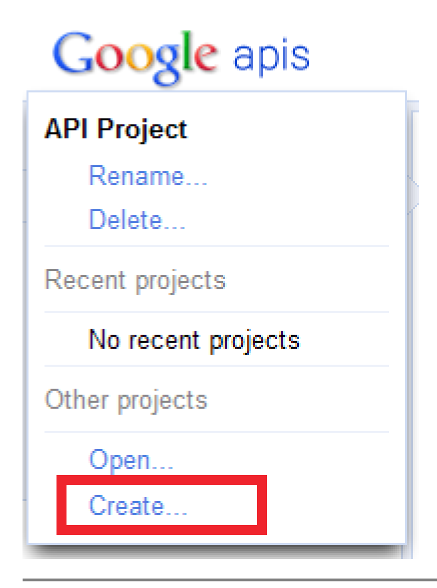
Show: Updates/New Installed Obsolete Select New or Updates Install 1 package...

Sort by: API level Repository Deselect All Delete packages...

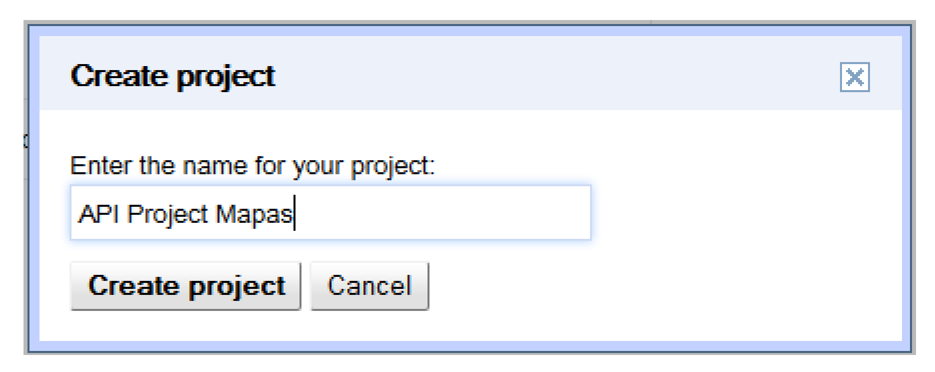
Done loading packages.

Tras pulsar el botón de Install y aceptar la licencia correspondiente el paquete quedará instalado en nuestro sistema, concretamente en la ruta: `<carpeta-sdk-android>/extras/google/google_play_services/`. Recordemos esto porque nos hará falta más adelante.

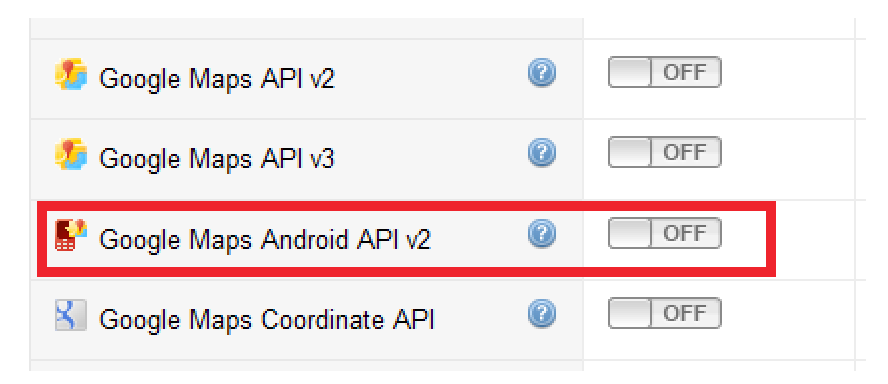
El siguiente paso será obtener una *API Key* para poder utilizar el servicio de mapas de Google en nuestra aplicación. Este paso ya lo comentamos en los artículos sobre la API v1, pero en este caso el procedimiento será algo distinto. La nueva API de mapas de Android se ha integrado por fin en la [Consola de APIs de Google], por lo que el primer paso será acceder a ella. Una vez hemos accedido, tendremos que crear un nuevo proyecto desplegando el menú superior izquierdo y seleccionando la opción "Create...".



Aparecerá entonces una ventana que nos solicitará el nombre del proyecto. Introducimos algún nombre descriptivo y aceptamos sin más.

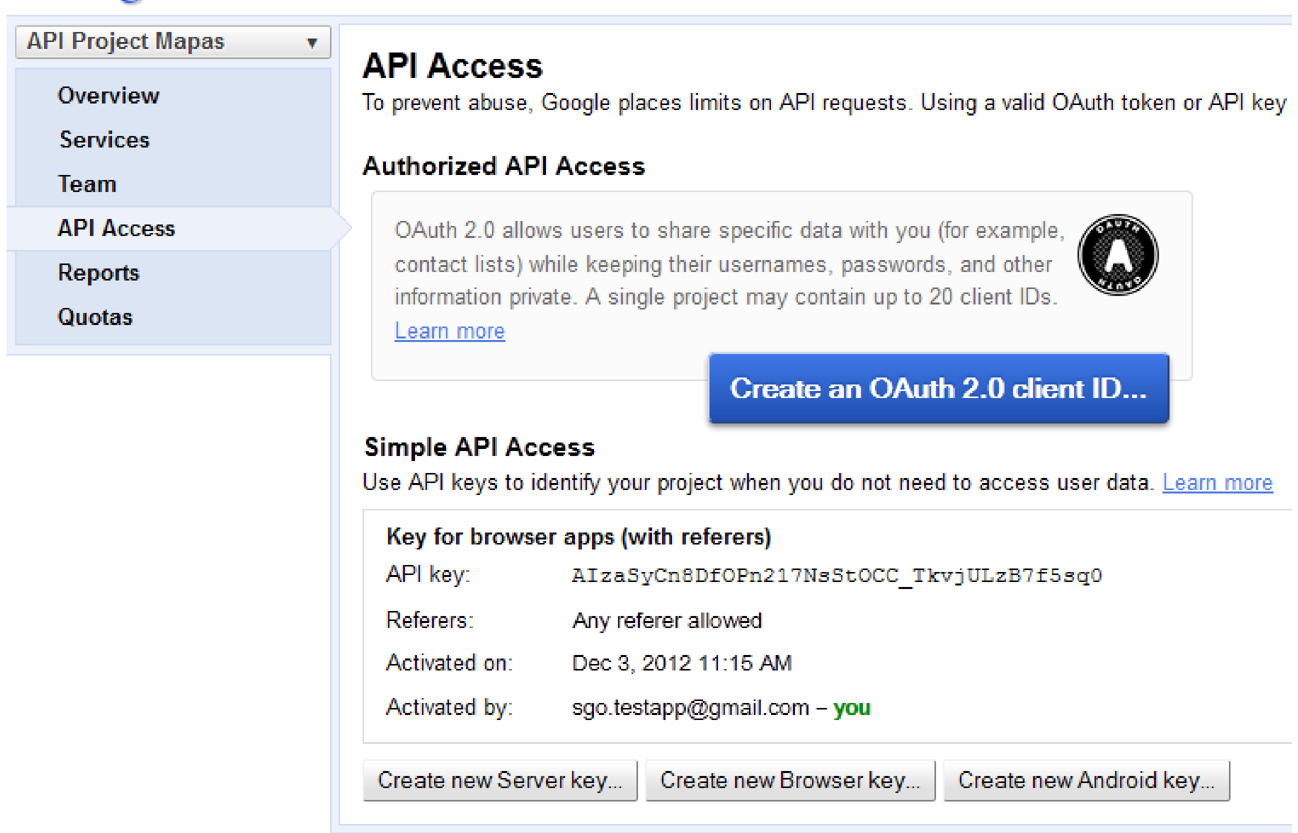


Una vez creado el proyecto, accederemos a la opción "Services" del menú izquierdo. Desde esta ventana podemos activar o desactivar cada uno de los servicios de Google que queremos utilizar. En este caso sólo activaremos el servicio llamado "Google Maps Android API v2" pulsando sobre el botón ON/OFF situado justo a su derecha.

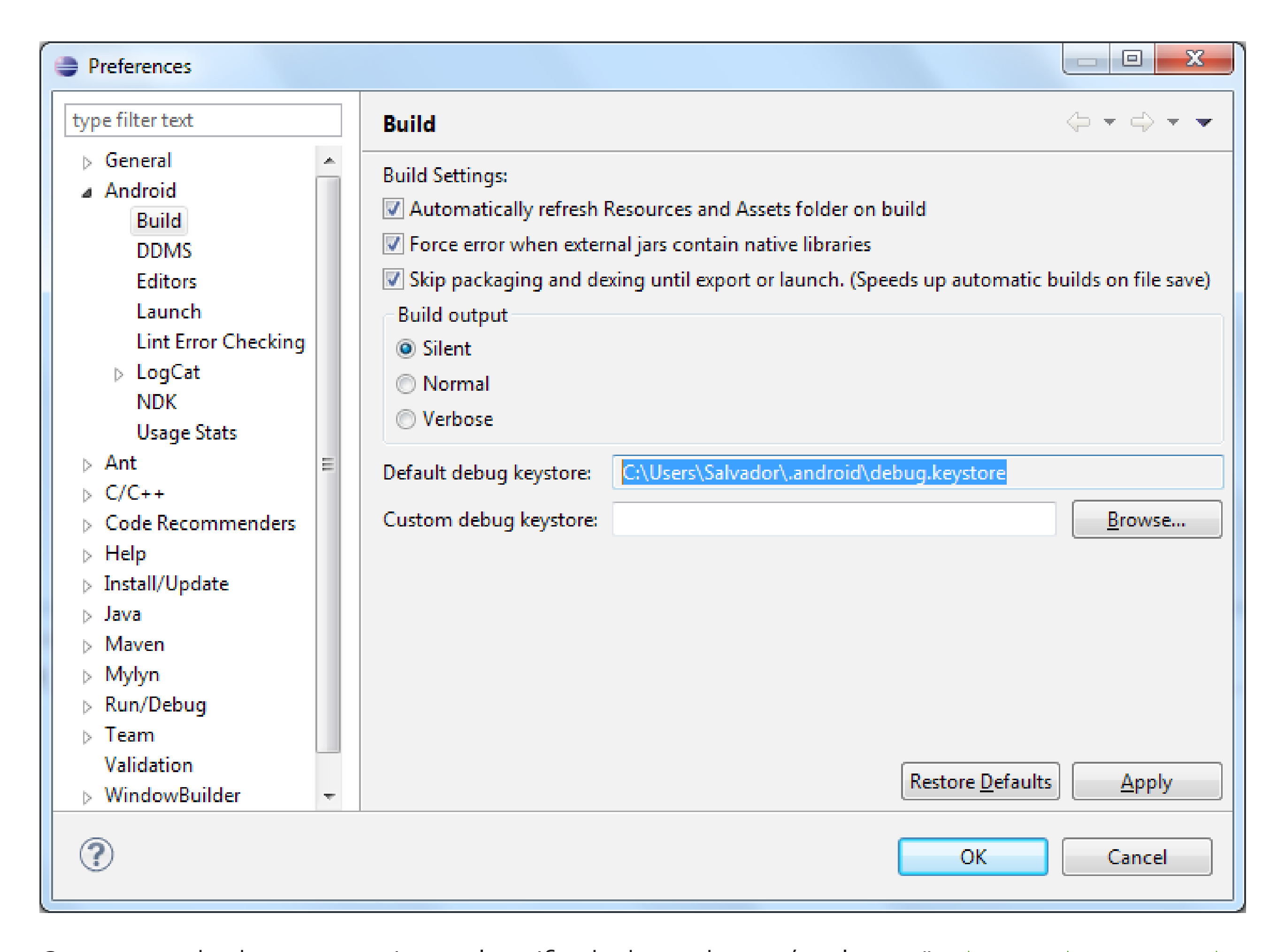


Una vez activado aparecerá una nueva opción en el menú izquierdo llamada "API Access". Accediendo a dicha opción tendremos la posibilidad de obtener nuestra nueva API Key que nos permita utilizar el servicio de mapas desde nuestra aplicación particular.





Para ello, pulsaremos el botón "Create new Android key...". Esto nos llevará a un cuadro de diálogo donde tendremos que introducir algunos datos identificativos de nuestra aplicación. En concreto necesitaremos dos: la huella digital (SHA1) del certificado con el que firmamos la aplicación, y el paquete java utilizado. El segundo no tiene misterio, pero el primero requiere alguna explicación. Toda aplicación Android debe ir firmada para poder ejecutarse en un dispositivo, tanto físico como emulado. Este proceso de firma es uno de los pasos que tenemos que hacer siempre antes de distribuir públicamente una aplicación. Adicionalmentes, durante el desarrollo de la misma, para realizar pruebas y la depuración del código, aunque no seamos conscientes de ello también estamos firmado la aplicación con un "certificado de pruebas". Podemos saber en qué carpeta de nuestro sistema está almacenado este certificado accediendo desde Eclipse al menú Window /Preferences y accediendo a la sección Android / Build.



Como se puede observar, en mi caso el certificado de pruebas está en la ruta "`C:\Users\Salvador\.android\debug.keystore`". Pues bien, para obtener nuestra huella digital SHA1 deberemos acceder a dicha ruta desde la consola de comando de Windows y ejecutar los siguientes comandos:

```
C:\>cd C:\Users\Salvador\.android\

C:\Users\Salvador\.android>"C:\Program Files\Java\jdk1.7.0_07\bin\keytool.exe"
-list -v -keystore debug.keystore -alias androiddebugkey -storepass android
-keypass android
```

Suponiendo que tu instalación de Java está en la ruta "`C:\Program Files\Java\jdk1.7.0_07`". Si no es así sólo debes sustituir ésta por la correcta. Esto nos deberá devolver varios datos del certificado, entre ellos la huella SHA1.

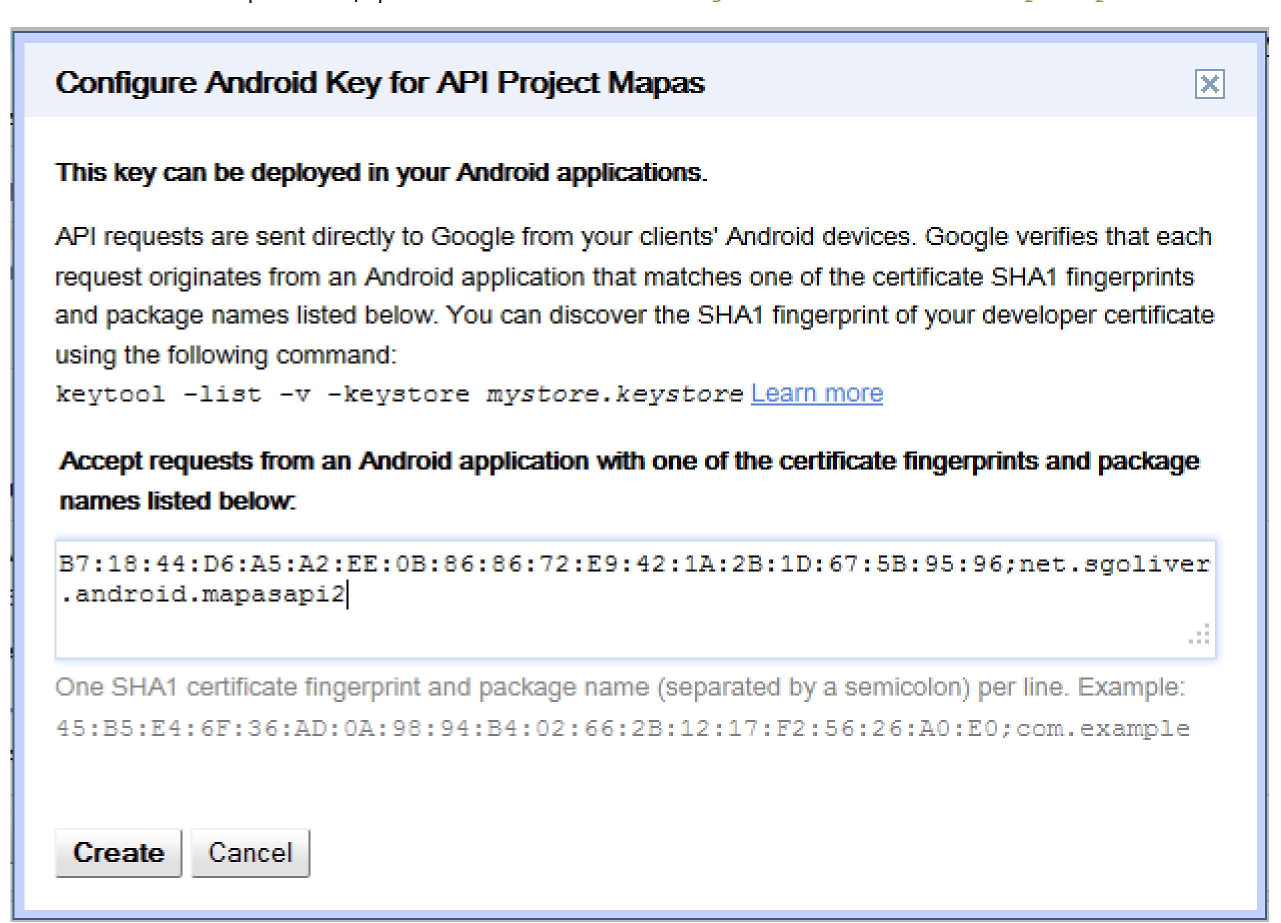
```
Seleccionar C:\Windows\system32\cmd.exe

C:\>cd C:\Users\Salvador\.android\

C:\Users\Salvador\.android>"C:\Program Files\Java\jdk1.7.0_07\bin\keytool.exe" -
list -v -keystore debug.keystore -alias androiddebugkey -storepass android -keyp
ass android
Nombre de Alias: androiddebugkey
Fecha de Creaci¾n: 30-nov-2011
Tipo de Entrada: PrivateKeyEntry
Longitud de la Cadena de Certificado: 1
Certificado[1]:
Propietario: CN=Android Debug, O=Android, C=US
Emisor: CN=Android Debug, O=Android, C=US
N·mero de serie: 4ed679f4
VßIido desde: Wed Nov 30 19:46:12 CET 2011 hasta: Fri Nov 22 19:46:12 CET 2041
Huellas digitales del Certificado:
         MD5: 5A:F0:7B:4F:75:88:BE:3C:B3:F0:52:0C:FA:94:4F:E6
         SHA1: B7:18:44:D6:A5:A2:EE:0B:86:86:72:E9:42:1A:2B:1D:67:5B:95:96
         SHA256: 50:EF:2E:51:0D:19:69:60:E3:2D:AE:4C:AE:EF:44:39:84:24:47:CA:C3:
71:D4:C8:9A:F2:8D:BC:97:09:2A:E9
         Nombre del Algoritmo de Firma: SHA1withRSA
         Versi¾n: 3

C:\Users\Salvador\.android>
```

Pues bien, nos copiamos este dato y lo añadimos a la ventana de obtención de la API Key donde nos habíamos quedado antes, y a continuación separado por un punto y coma añadimos el paquete java que vayamos a utilizar en nuestra aplicación, que en mi caso será "`net.sgoliver.android.mapasapi2`".



Pulsamos el botón "Create" y ya deberíamos tener nuestra API Key generada, podremos verla en la pantalla siguiente dentro del apartado "Key for Android Apps (with certificates)". Apuntaremos también este dato para utilizarlo más tarde.

Simple API Access

Use API keys to identify your project when you do not need to access user data. Learn more

Key for Android apps (with certificates)

API key:	AIzaSyCZXCc8_tA_vVo7d2aOqn9LOWLa1cAR0fQ
Android apps:	B7:18:44:D6:A5:A2:EE:0B:86:86:72:E9:42:1A:2B:1D:67:5I
Activated on:	Dec 3, 2012 11:18 AM
Activated by:	sgo.testapp@gmail.com – **you**

Con esto ya habríamos concluido los preparativos iniciales necesarios para utilizar el servicio de mapas de Android en nuestras propias aplicaciones, por lo que empecemos a crear un proyecto de ejemplo en Eclipse.

Abriremos Eclipse y crearemos un nuevo proyecto estandar de Android, en mi caso lo he llamado "android-mapas-api2". Recordemos utilizar para el proyecto el mismo paquete java que hemos indicado durante la obtención de la API key.

Tras esto lo primero que haremos será añadir al fichero AndroidManifest.xml la API Key que acabamos de generar. Para ello añadiremos al fichero, dentro de la etiqueta <application>, un nuevo elemento <meta-data> con los siguientes datos:

```
...
<application>
...
    <meta-data android:name="com.google.android.maps.v2.API_KEY"
               android:value="api_key"/>
...
</application>
```

Como valor del parámetro android:value tendremos que poner nuestra API Key recien generada.

Siguiendo con el AndroidManifest, también tendremos que incluir una serie de permisos que nos permitan hacer uso de los mapas. En primer lugar tendremos que definir y utilizar un permiso llamado "tu.paquete.java.permission.MAPS_RECEIVE", en mi caso quedaría de la siguiente forma:

```
<permission
      android:name="net.sgoliver.android.mapasapi2.permission.MAPS_RECEIVE"
      android:protectionLevel="signature"/>

<uses-permission android:name="net.sgoliver.android.mapasapi2.permission.MAPS_
RECEIVE"/>
```

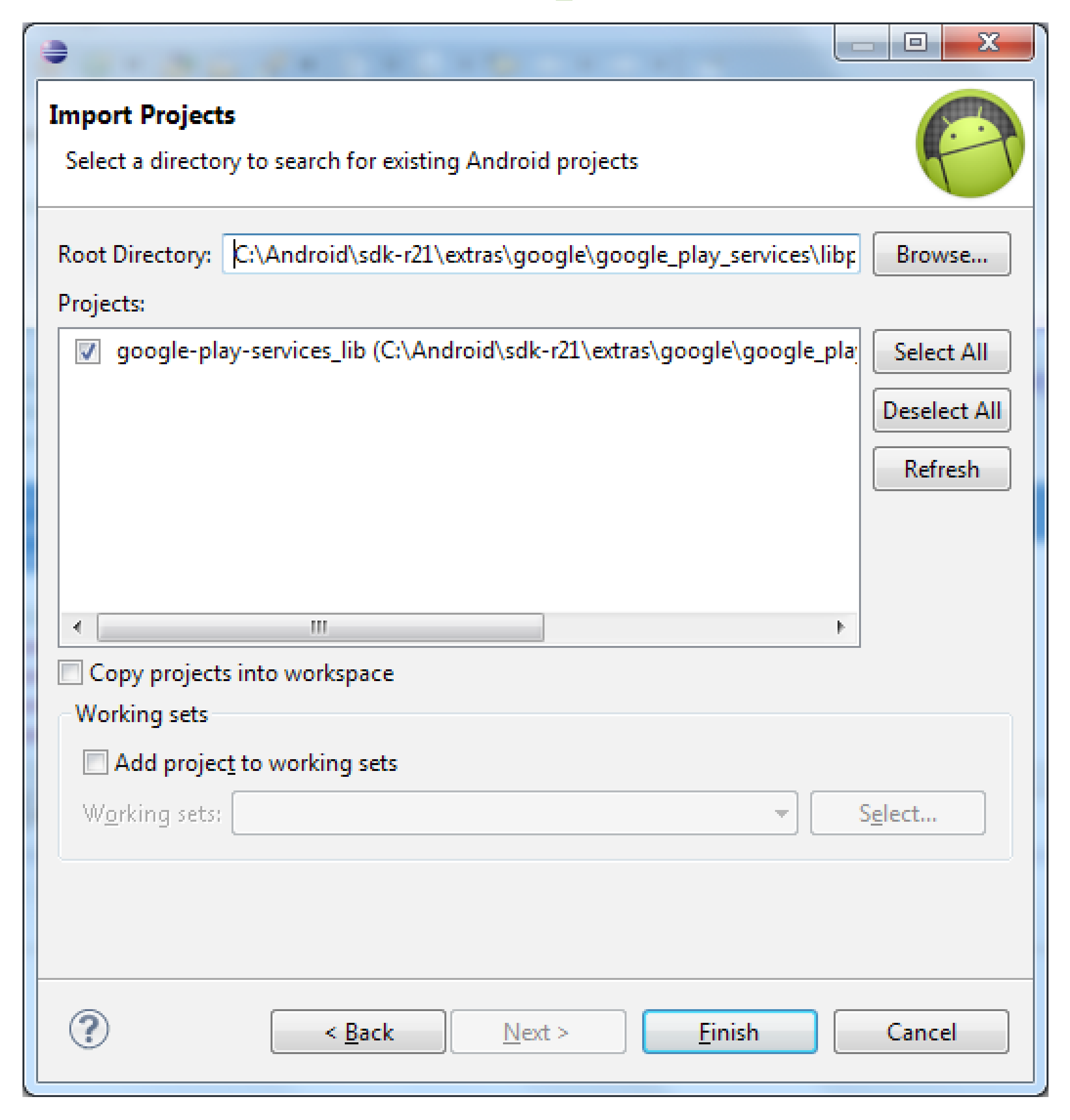
Además, tendremos que añadir permisos adicionales que nos permitan acceder a los servicios web de Google, a Internet, y al almacenamiento externo del dispositivo (utilizado para la caché de los mapas):

```
<uses-permission android:name="com.google.android.providers.gsf.permission.
READ_GSERVICES"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Por último, dado que la API v2 de Google Maps Android utiliza OpenGL ES versión 2, deberemos especificar también dicho requisito en nuestro AndroidManifest añadiendo un nuevo elemento <uses-feature>:

```
<uses-feature android:glEsVersion="0x00020000"
              android:required="true"/>
```

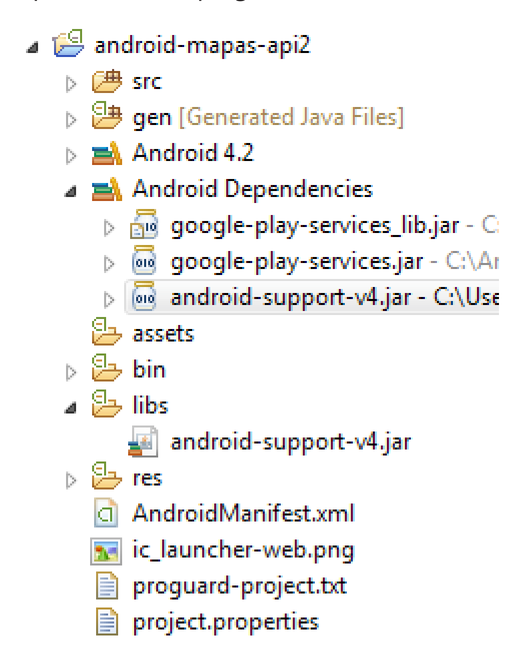
Una vez hemos configurado todo lo necesario en el `AndroidManifest`, y antes de escribir nuestro código, tenemos que seguir añadiendo elementos externos a nuestro proyecto. El primero de ellos será referenciar desde nuestro proyecto la librería con el SDK de Google Play Services que nos descargamos al principio de este tutorial. Para ello, desde Eclipse podemos importar la librería a nuestro conjunto de proyectos mediante la opción de menú "File / Import... / Existing Android Code Into Workspace". Como ya dijimos este paquete se localiza en la ruta "`<carpeta-sdk-android>/extras/google/google_play_services/libproject/google-play-services_lib`".



Tras seleccionar la ruta correcta dejaremos el resto de opciones con sus valores por defecto y pulsaremos Finish para que Eclipse importe esta librería a nuestro conjunto de proyectos.

El siguiente paso será referenciar esta librería desde nuestro proyecto de ejemplo. Para ello iremos a sus propiedades pulsando botón derecho / Properties sobre nuestro proyecto y accediendo a la sección Android de las preferencias. En dicha ventana podemos añadir una nueva librería en la sección inferior llamada Library. Cuando pulsamos el botón "Add..." nos aparecerá la librería recien importada y podremos seleccionarla directamente, añadiéndose a nuestra lista de librerías referenciadas por nuestro proyecto.

Como último paso de configuración de nuestro proyecto, si queremos que nuestra aplicación se pueda ejecutar desde versiones "antiguas" de Android (concretamente desde la versión de Android 2.2) deberemos asegurarnos de que nuestro proyecto incluye la librería `android-support-v4.jar`, que debería aparecer si desplegamos las sección "Android Dependencies" o la carpeta "`lib`" de nuestro proyecto.



Las versiones más recientes de ADT incluyen por defecto esta librería en nuestros proyectos, pero si no está incluida podéis hacerlo mediante la opción del menú contextual "Android Tools / Add Support Library..." sobre el proyecto, o bien de forma manual.

Y con esto hemos terminado de configurar todo lo necesario. Ya podemos escribir nuestro código. Y para este primer apartado sobre el tema nos vamos a limitar a mostrar un mapa en la pantalla principal de la aplicación. En apartados posteriores veremos como añadir otras opciones o elementos al mapa.

Para esto tendremos simplemente que añadir el control correspondiente al layout de nuestra actividad principal. En el caso de la nueva API v2 este "control" se añadirá en forma de *fragment* (de ahí que hayamos tenido que incluir la librería android-support para poder utilizarlos en versiones de Android anteriores a la 3.0) de un determinado tipo (concretamente de la nueva clase `com.google.android.gms.maps.SupportMapFragment`), quedando por ejemplo de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/map"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.google.android.gms.maps.SupportMapFragment"/>
```

Por supuesto, dado que estamos utilizando fragments, la actividad principal también tendrá que extender a `FragmentActivity` (en vez de simplemente `Activity` como es lo "normal"). Usaremos también la versión de `FragmentActivity` incluida en la librería `android-support` para ser compatibles con la mayoría de las versiones Android actuales.

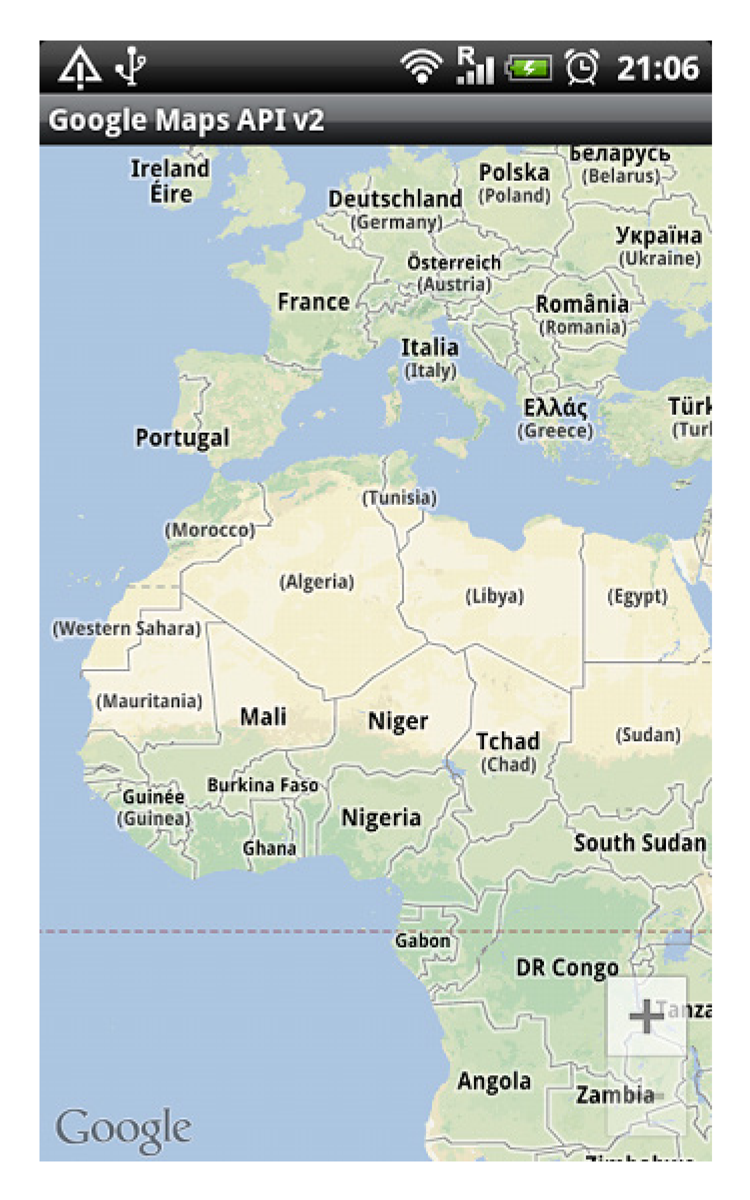
```
public class MainActivity extends android.support.v4.app.FragmentActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    }

...
}
```

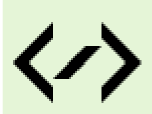
Con esto, ya podríamos ejecutar y probar nuestra aplicación. En mi caso las pruebas las he realizado sobre un dispositivo físico con Android 2.2 ya que por el momento parece haber algunos problemas para hacerlo sobre el emulador. Por tanto tendréis que conectar vuestro dispositivo al PC mediante el cable de datos e indicar a Eclipse que lo utilice para la ejecución de la aplicación.

Si ejecutamos el ejemplo deberíamos ver un mapa en la pantalla principal de la aplicación, sobre el que nos podremos mover y hacer zoom con los gestos habituales o utilizando los controles de zoom incluidos por defecto sobre el mapa.



Con este capítulo espero haber descrito todos los pasos necesarios para comenzar a utilizar los servicios de mapas de Google utilizando su nueva API Google Maps Android v2.

Como habéis podido comprobar hay muchos preparativos que hacer, aunque ninguno de ellos de excesiva dificultad. En los próximos apartados aprenderemos a utilizar más características de la nueva API.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-mapas-api2

Opciones generales del mapa

En el apartado anterior hemos visto cómo realizar todos los preparativos necesarios para comenzar a utilizar la nueva versión de Google Maps para Android (Google Maps Android API v2): descargar las librerías necesarias, obtener la API Key y configurar un nuevo proyecto en Eclipse.

En esta segunda entrega vamos a hacer un repaso de las opciones básicas de los nuevos mapas: elegir el tipo de mapa a mostrar, movernos por él de forma programática, y obtener los datos de la posición actual. Como aplicación de ejemplo tomaremos como base la ya creada en el apartado anterior, a la que añadiremos

varias opciones de menú para demostrar el funcionamiento de algunas funciones del mapa.

Si hacemos un poco de memoria, recordaremos cómo en la antigua versión de la API de Google Maps era bastante poco homogéneo el acceso y modificación de determinados datos del mapa. Por ejemplo, la consulta de la posición actual o la configuración del tipo de mapa se hacían directamente sobre el control `MapView`, mientras que la manipulación de la posición y el zoom se hacían a través del controlador asociado al mapa (`MapController`). Además, el tratamiento de las coordenadas y las unidades utilizadas eran algo peculiares, teniendo estar continuamente convirtiendo de grados a microgrados y de estos a objetos `GeoPoint`, etc.

Con la nueva API, todas las operaciones se realizarán directamente sobre un objeto `GoogleMap`, el componente base de la API. Accederemos a este componente llamando al método `getMap()` del fragmento `MapFragment` que contenga nuestro mapa. Podríamos hacerlo de la siguiente forma:

```
import com.google.android.gms.maps.GoogleMap;
...
GoogleMap mapa = ((SupportMapFragment) getSupportFragmentManager()
                          .findFragmentById(R.id.map)).getMap();
```

Una vez obtenida esta referencia a nuestro objeto `GoogleMap` podremos realizar sobre él la mayoría de las acciones básicas del mapa.

Así, por ejemplo, para modificar el tipo de mapa mostrado podremos utilizar una llamada a su método `setMapType()`, pasando como parámetro el tipo de mapa:

- `MAP_TYPE_NORMAL`.
- `MAP_TYPE_HYBRID`.
- `MAP_TYPE_SATELLITE`.
- `MAP_TYPE_TERRAIN`.

Para nuestro ejemplo voy a utilizar una variable que almacene el tipo de mapa actual (del 0 al 3) y habilitaremos una opción de menú para ir alternando entre las distintas opciones. Quedaría de la siguiente forma:

```
private void alternarVista()
{
    vista = (vista + 1) % 4;

    switch(vista)
    {
        case 0:
            mapa.setMapType(GoogleMap.MAP_TYPE_NORMAL);
            break;
        case 1:
            mapa.setMapType(GoogleMap.MAP_TYPE_HYBRID);
            break;
        case 2:
            mapa.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
            break;
        case 3:
            mapa.setMapType(GoogleMap.MAP_TYPE_TERRAIN);
            break;
    }
}
```

En cuanto al movimiento sobre el mapa, con esta nueva versión de la API vamos a tener mucha más libertad

que con la anterior versión, ya que podremos mover libremente nuestro punto de vista (o cámara, como lo han llamado los chicos de Android) por un espacio 3D. De esta forma, ya no sólo podremos hablar de latitud-longitud (*target*) y zoom, sino también de orientación (*bearing*) y ángulo de visión (*tilt*). La manipulación de los 2 últimos parámetros unida a posibilidad actual de ver edificios en 3D de muchas ciudades nos abren un mundo de posibilidades.

El movimiento de la cámara se va a realizar siempre mediante la construcción de un objeto `CameraUpdate` con los parámetros necesarios. Para los movimientos más básicos como la actualización de la latitud y longitud o el nivel de zoom podremos utilizar la clase `CameraUpdateFactory` y sus métodos estáticos que nos facilitará un poco el trabajo.

Así por ejemplo, para cambiar sólo el nivel de zoom podremos utilizar los siguientes métodos para crear nuestro `CameraUpdate`:

- `CameraUpdateFactory.zoomIn()`. Aumenta en 1 el nivel de zoom.
- `CameraUpdateFactory.zoomOut()`. Disminuye en 1 el nivel de zoom.
- `CameraUpdateFactory.zoomTo(nivel_de_zoom)`. Establece el nivel de zoom.

Por su parte, para actualizar sólo la latitud-longitud de la cámara podremos utilizar:

- `CameraUpdateFactory.newLatLng(lat, long)`. Establece la lat-lng expresadas en grados.

Si queremos modificar los dos parámetros anteriores de forma conjunta, también tendremos disponible el método siguiente:

- `CameraUpdateFactory.newLatLngZoom(lat, long, zoom)`. Establece la lat-lng y el zoom.

Para movernos lateralmente por el mapa (*panning*) podríamos utilizar los métodos de scroll:

- `CameraUpdateFactory.scrollBy(scrollHorizontal, scrollVertical)`. Scroll expresado en píxeles.

Tras construir el objeto `CameraUpdate` con los parámetros de posición tendremos que llamar a los métodos `moveCamera()` o `animateCamera()` de nuestro objeto `GoogleMap`, dependiendo de si queremos que la actualización de la vista se muestre directamente o de forma animada.

Con esto en cuenta, si quisiéramos por ejemplo centrar la vista en España con un zoom de 5 podríamos hacer lo siguiente:

```
CameraUpdate camUpd1 =
    CameraUpdateFactory.newLatLng(new LatLng(40.41, -3.69));

mapa.moveCamera(camUpd1);
```

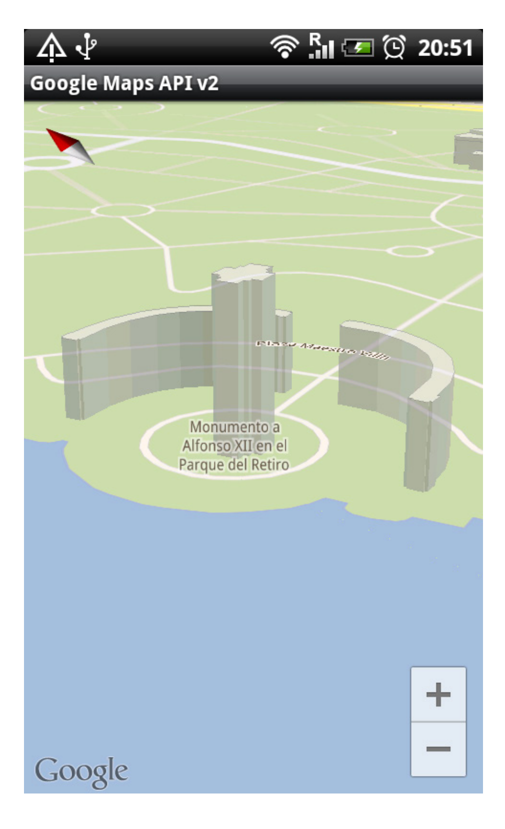
Además de los movimientos básicos que hemos comentado, si queremos modificar los demás parámetros de la cámara o varios de ellos simultaneamente tendremos disponible el método más general `CameraUpdateFactory.newCameraPosition()` que recibe como parámetro un objeto de tipo `CameraPosition`. Este objeto los construiremos indicando todos los parámetros de la posición de la cámara a través de su método `Builder()` de la siguiente forma:

```
LatLng madrid = new LatLng(40.417325, -3.683081);
CameraPosition camPos = new CameraPosition.Builder()
        .target(madrid)   //Centramos el mapa en Madrid
        .zoom(19)         //Establecemos el zoom en 19
        .bearing(45)      //Establecemos la orientación con el noreste arriba
        .tilt(70)         //Bajamos el punto de vista de la cámara 70 grados
        .build();

CameraUpdate camUpd3 =
    CameraUpdateFactory.newCameraPosition(camPos);

mapa.animateCamera(camUpd3);
```

Como podemos comprobar, mediante este mecanismo podemos modificar todos los parámetros de posición de la cámara (o sólo algunos de ellos) al mismo tiempo. En nuestro caso de ejemplo hemos centrado la vista del mapa sobre el parque de *El Retiro* de Madrid, con un nivel de zoom de 19, una orientación de 45 grados para que el noreste esté hacia arriba y un ángulo de visión de 70 grados de forma que veamos en 3D el monumento a Alfonso XII en la vista de mapa `NORMAL`. En la siguiente imagen vemos el resultado:



Como podéis ver, en esta nueva versión de la API se facilita bastante el posicionamiento dentro del mapa, y el uso de las clases `CameraUpdate` y `CameraPosition` resulta bastante intuitivo.

Bien, pues ya hemos hablado de cómo modificar nuestro punto de vista sobre el mapa, pero si el usuario se mueve de forma manual por él, ¿cómo podemos conocer en un momento dado la posición de la cámara?

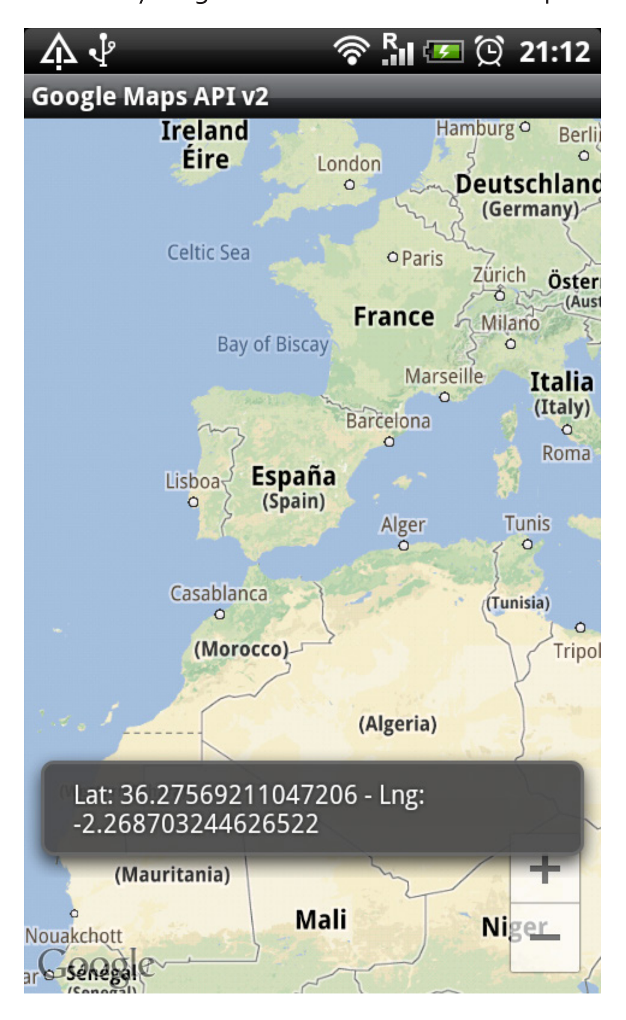
Pues igual de fácil, mediante el método `getCameraPosition()`, que nos devuelve un objeto `CameraPosition` como el que ya conocíamos. Accediendo a los distintos métodos y propiedades de este objeto podemos conocer con exactitud la posición de la cámara, la orientación y el nivel de zoom.

```
CameraPosition camPos = mapa.getCameraPosition();

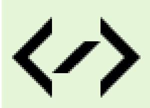
LatLng coordenadas = camPos.target;
double latitud = coordenadas.latitude;
double longitud = coordenadas.longitude;

float zoom = camPos.zoom;
float orientacion = camPos.bearing;
float angulo = camPos.titl;
```

En nuestra aplicación de ejemplo he añadido una nueva opción de menú que muestra en un mensaje toast la latitud y longitud actual de la vista de mapa.



Y con esto habríamos terminado de describir las acciones básicas de configuración y movimiento sobre el mapa. En los próximos apartados veremos más opciones, como la forma de añadir marcadores o dibujar sobre el mapa.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-mapas-api2-p2

Eventos, marcadores y dibujo sobre el mapa

En los dos apartados anteriores hemos visto cómo crear aplicaciones utilizando la nueva versión de la API v2 de Google Maps para Android y hemos descrito las acciones principales sobre el mapa.

En este último apartado de la serie nos vamos a centrar en los eventos del mapa que podemos capturar y tratar, en la creación y gestión de marcadores, y en el dibujo de lineas y polígonos sobre el mapa.

Sin más preámbulos, comencemos con los eventos. Si hacemos un poco de memoria, con la versión anterior de la API debíamos crear una nueva capa (*overlay*) para capturar los eventos principales de pulsación. Sin embargo, el nuevo componente de mapas soporta directamente los eventos de *click*, *click largo* y *movimiento de cámara* y podemos implementarlos de la forma habitual, mediante su método *set* correspondiente.

Así por ejemplo, podríamos implementar el evento de onclick llamando al método `setOnMapClickListener()` con un nuevo listener y sobrescribir el método `onMapClick()`. Este método recibe como parámetro, en forma de objeto `LatLng`, las coordenadas de latitud y longitud sobre las que ha pulsado el usuario. Si quisiéramos traducir estas coordenadas física en coordenadas en pantalla podríamos utilizar un objeto `Projection` (similar al que ya comentamos para la API v1), obteniéndolo a partir del mapa a través del método `getProjection()` y posteriormente llamando a `toScreenLocation()` para obtener las coordenadas (x,y) de pantalla donde el usuario pulsó.

Así, por ejemplo, si quisiéramos mostrar un `Toast` con todos estos datos cada vez que se pulse sobre el mapa podríamos hacer lo siguiente:

```
mapa.setOnMapClickListener(new OnMapClickListener() {
    public void onMapClick(LatLng point) {
        Projection proj = mapa.getProjection();
        Point coord = proj.toScreenLocation(point);

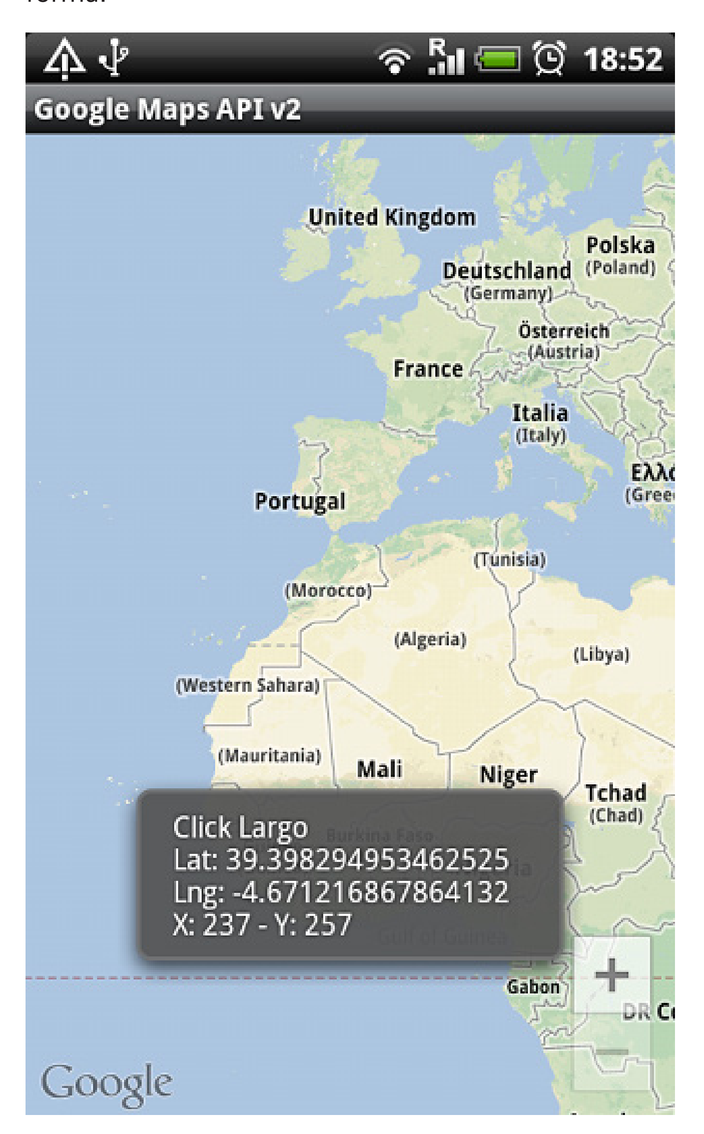
        Toast.makeText(
            MainActivity.this,
            "Click\n" +
            "Lat: " + point.latitude + "\n" +
            "Lng: " + point.longitude + "\n" +
            "X: " + coord.x + " - Y: " + coord.y,
            Toast.LENGTH_SHORT).show();
    }
});
```

De forma similar podríamos implementar el evento de pulsación larga, con la única diferencia de que lo asignaríamos mediante `setOnMapLongClickListener()` y sobrescribiríamos el método `onMapLongClick()`.

```
mapa.setOnMapLongClickListener(new OnMapLongClickListener() {
    public void onMapLongClick(LatLng point) {
        Projection proj = mapa.getProjection();
        Point coord = proj.toScreenLocation(point);

        Toast.makeText(
            MainActivity.this,
            "Click Largo\n" +
            "Lat: " + point.latitude + "\n" +
            "Lng: " + point.longitude + "\n" +
            "X: " + coord.x + " - Y: " + coord.y,
            Toast.LENGTH_SHORT).show();
    }
});
```

Así, cuando el usuario hiciera una pulsación larga sobre el mapa veríamos los datos en pantalla de la siguiente forma:



También podremos capturar el evento de cambio de cámara, de forma que podamos realizar determinadas acciones cada vez que el usuario se mueve manualmente por el mapa, desplazándolo, haciendo zoom, o modificando la orientación o el ángulo de visión. Este evento lo asignaremos al mapa mediante su método `setOnCameraChangeListener()` y sobrescribiendo el método `onCameraChange()`. Este método recibe como parámetro un objeto `CameraPosition`, que ya vimos en el apartado anterior, por lo que podremos recuperar de él todos los datos de la cámara en cualquier momento.

De esta forma, si quisiéramos mostrar un `Toast` con todos los datos podríamos hacer lo siguiente:

```
mapa.setOnCameraChangeListener(new OnCameraChangeListener() {
    public void onCameraChange(CameraPosition position) {
        Toast.makeText(
            MainActivity.this,
            "Cambio Cámara\n" +
            "Lat: " + position.target.latitude + "\n" +
            "Lng: " + position.target.longitude + "\n" +
            "Zoom: " + position.zoom + "\n" +
            "Orientación: " + position.bearing + "\n" +
            "Ángulo: " + position.tilt,
            Toast.LENGTH_SHORT).show();
    }
});
```

Hecho esto, cada vez que el usuario se mueva por el mapa veríamos lo siguiente:



El siguiente tema importante que quería tratar en este apartado es el de los marcadores. Rara es la aplicación Android que hace uso de mapas sin utilizar también este tipo de elementos para resaltar determinados puntos en el mapa. Si recordamos los artículos sobre la API v1, vimos cómo podíamos añadir marcadores añadiendo una nueva capa (*overlay*) al mapa y dibujando nuestro marcador como parte de su evento `draw()`. En la nueva versión de la API tendemos toda esta funcionalidad integrada en la propia vista de mapa, y agregar un marcador resulta tan sencillo como llamar al método `addMarker()` pasándole la posición en forma de objeto `LatLng` y el texto a mostrar en la ventana de información del marcador. En nuestra aplicación de ejemplo añadiremos un menú de forma que cuando lo pulsemos se añada automáticamente un marcador

sobre España con el texto "*Pais: España*". Veamos cómo escribir un método auxiliar que nos ayuda a hacer esto pasándole las coordenadas de latitud y longitud:

```
private void mostrarMarcador(double lat, double lng)
{
    mapa.addMarker(new MarkerOptions()
        .position(new LatLng(lat, lng))
        .title("Pais: España"));
}
```

Así de sencillo, basta con llamar al método `addMarker()` pasando como parámetro un nuevo objeto `MarkerOptions` sobre el que establecemos la posición del marcador (método `position()`) y el texto a incluir en la ventana de información del marcador (métodos `title()` para el título y `snippet()` para el resto del texto). Si ejecutamos la aplicación de ejemplo y pulsamos el menú "Marcadores" aparecerá el siguiente marcador sobre el mapa (la ventana de información aparece si además se pulsa sobre el marcador):



Además de facilitarnos la vida con la inclusión de marcadores en el mapa también tendremos ayuda a la hora de capturar el evento de pulsación sobre un marcador, ya que podremos asignar al mapa dicho evento como cualquiera de los comentados anteriormente. En este caso el evento se asignará al mapa mediante el método `setOnMarkerClickListener()` y sobrescribiremos el método `onMarkerClick()`. Dicho método recibe como parámetro el objeto `Marker` pulsado, de forma que podamos identificarlo accediendo a su información (posición, título, texto, …). Veamos un ejemplo donde mostramos un toast con el título del marcador pulsado:

```
mapa.setOnMarkerClickListener(new OnMarkerClickListener() {
    public boolean onMarkerClick(Marker marker) {
        Toast.makeText(
            MainActivity.this,
            "Marcador pulsado:\n" +
            marker.getTitle(),
            Toast.LENGTH_SHORT).show();

        return false;
    }
});
```

Con el código anterior, si pulsamos sobre el marcador de España aparecerá el siguiente mensaje informativo:



Todo lo explicado sobre marcadores corresponde al comportamiento por defecto de la API, sin embargo también es posible por supuesto personalizar determinadas cosas, como por ejemplo el aspecto de los marcadores. Esto se sale un poco de este capítulo, donde pretendía describir los temas más básicos, pero para quien esté interesado tan sólo decir que mediante los métodos `icon()` y `anchor()` del objeto `MakerOptions` que hemos visto antes es posible utilizar una imagen personalizada para mostrar como marcador en el mapa. En la documentación oficial (en inglés) podéis encontrar un ejemplo de cómo hacer esto.

Como último tema, vamos a ver cómo dibujar líneas y polígonos sobre el mapa, elementos muy comunmente utilizados para trazar rutas o delimitar zonas del mapa. Para realizar esto en la versión 2 de la API vamos a

actuar una vez más directamente sobre la vista de mapa, sin necesidad de añadir *overlays* o similares, y ayudándonos de los objetos `PolylineOptions` y `PolygonOptions` respectivamente.

Para dibujar una linea lo primero que tendremos que hacer será crear un nuevo objeto PolylineOptions sobre el que añadiremos utilizando su método `add()` las coordenadas (latitud-longitud) de todos los puntos que conformen la linea. Tras esto estableceremos el grosor y color de la linea llamando a los métodos `width()` y `color()` respectivamente, y por último añadiremos la linea al mapa mediante su método `addPolyline()` pasándole el objeto `PolylineOptions` recién creado.

En nuestra aplicación de ejemplo he añadido un nuevo menú para dibujar un rectángulo sobre España. Veamos cómo queda:

```
private void mostrarLineas()
{
    //Dibujo con Lineas

    PolylineOptions lineas = new PolylineOptions()
            .add(new LatLng(45.0, -12.0))
            .add(new LatLng(45.0, 5.0))
            .add(new LatLng(34.5, 5.0))
            .add(new LatLng(34.5, -12.0))
            .add(new LatLng(45.0, -12.0));

    lineas.width(8);
    lineas.color(Color.RED);

    mapa.addPolyline(lineas);
}
```

Ejecutando esta acción en el emulador veríamos lo siguiente:



Pues bien, esto mismo podríamos haberlo logrado mediante el dibujo de polígonos, cuyo funcionamiento es muy similar. Para ello crearíamos un nuevo objeto `PolygonOptions` y añadiremos las coordenadas de sus puntos en el sentido de las agujas del reloj. En este caso no es necesario cerrar el circuito (es decir, que la primera coordenada y la última fueran iguales) ya que se hace de forma automática. Otra diferencia es que para polígonos el ancho y color de la linea los estableceríamos mediante los métodos `strokeWidth()` y `strokeColor()`. Además, el dibujo final del polígono sobre el mapa lo haríamos mediante `addPolygon()`. En nuestro caso quedaría como sigue:

```
//Dibujo con polígonos

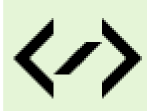
PolygonOptions rectangulo = new PolygonOptions()
              .add(new LatLng(45.0, -12.0),
                   new LatLng(45.0, 5.0),
                   new LatLng(34.5, 5.0),
                   new LatLng(34.5, -12.0),
                   new LatLng(45.0, -12.0));

rectangulo.strokeWidth(8);
rectangulo.strokeColor(Color.RED);

mapa.addPolygon(rectangulo);
```

El resultado al ejecutar la acción en el emulador debería ser exactamente igual que el anterior.

Y con esto habríamos concluido el capítulo destinado a describir el funcionamiento básico de la nueva API v2 de Google Maps para Android. Espero haber aclarado las dudas principales a la hora de comenzar a utilizar la nueva API.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-mapas-api2-p3

10

Ficheros

X. Ficheros en Android

Ficheros en Memoria Interna

En apartados anteriores hemos visto ya diversos métodos para almacenar datos en nuestras aplicaciones, como por ejemplo los ficheros de preferencias compartidas o las bases de datos SQLite. Estos mecanismos son perfectos para almacenar datos estructurados, pero en ocasiones nos seguirá siendo útil poder disponer también de otros ficheros auxiliares de datos, probablemente con otro tipo de contenidos y formatos. Por ello, en Android también podremos manipular ficheros tradicionales de una forma muy similar a como se realiza en Java.

Lo primero que hay que tener en cuenta es dónde queremos almacenar los ficheros y el tipo de acceso que queremos tener a ellos. Así, podremos leer y escribir ficheros localizados en:

- La **memoria interna** del dispositivo.
- La **tarjeta SD** externa, si existe.
- La propia aplicación, en forma de **recurso**.

En los dos próximos apartados aprenderemos a manipular ficheros almacenados en cualquiera de estos lugares, comentando las particularidades de cada caso.

Veamos en primer lugar cómo trabajar con la memoria interna del dispositivo. Cuando almacenamos ficheros en la memoria interna debemos tener en cuenta las limitaciones de espacio que tienen muchos dispositivos, por lo que no deberíamos abusar de este espacio utilizando ficheros de gran tamaño.

Escribir ficheros en la memoria interna es muy sencillo. Android proporciona para ello el método `openFileOutput()`, que recibe como parámetros el nombre del fichero y el modo de acceso con el que queremos abrir el fichero. Este modo de acceso puede variar entre `MODE_PRIVATE` para acceso privado desde nuestra aplicación (crea el fichero o lo sobrescribe si ya existe), `MODE_APPEND` para añadir datos a un fichero ya existente, `MODE_WORLD_READABLE` para permitir a otras aplicaciones leer el fichero, o `MODE_WORLD_WRITABLE` para permitir a otras aplicaciones escribir sobre el fichero. Los dos últimos no deberían utilizarse dada su peligrosidad, de hecho, han sido declarados como obsoletos (*deprecated*) en la API 17.

Este método devuelve una referencia al *stream* de salida asociado al fichero (en forma de objeto `FileOutputStream`), a partir del cual ya podremos utilizar los métodos de manipulación de ficheros tradicionales del lenguaje java (api `java.io`). Como ejemplo, convertiremos este *stream* a un `OutputStreamWriter` para escribir una cadena de texto al fichero.

```
try
{
    OutputStreamWriter fout=
        new OutputStreamWriter(
            openFileOutput("prueba_int.txt", Context.MODE_PRIVATE));

    fout.write("Texto de prueba.");
    fout.close();
}
catch (Exception ex)
{
    Log.e("Ficheros", "Error al escribir fichero a memoria interna");
}
```

Está bien, ya hemos creado un fichero de texto en la memoria interna, ¿pero dónde exactamente? Tal como ocurría con las bases de datos SQLite, Android almacena por defecto los ficheros creados en una ruta

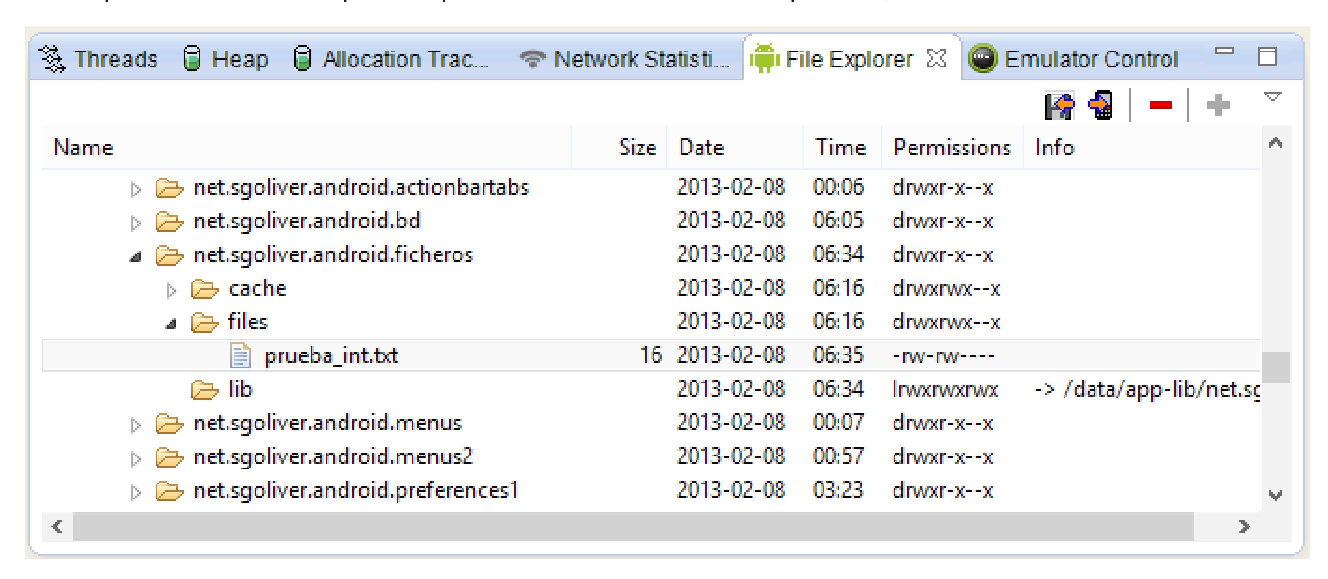
determinada, que en este caso seguirá el siguiente patrón:

```
/data/data/paquete_java/files/nombre_fichero
```

En mi caso particular, la ruta será

```
/data/data/net.sgoliver.android.ficheros/files/prueba_int.txt
```

Si ejecutamos el código anterior podremos comprobar en el DDMS cómo el fichero se crea correctamente en la ruta indicada (Al final del apartado hay un enlace a una aplicación de ejemplo sencilla donde incluyo un botón por cada uno de los puntos que vamos a comentar en el apartado).



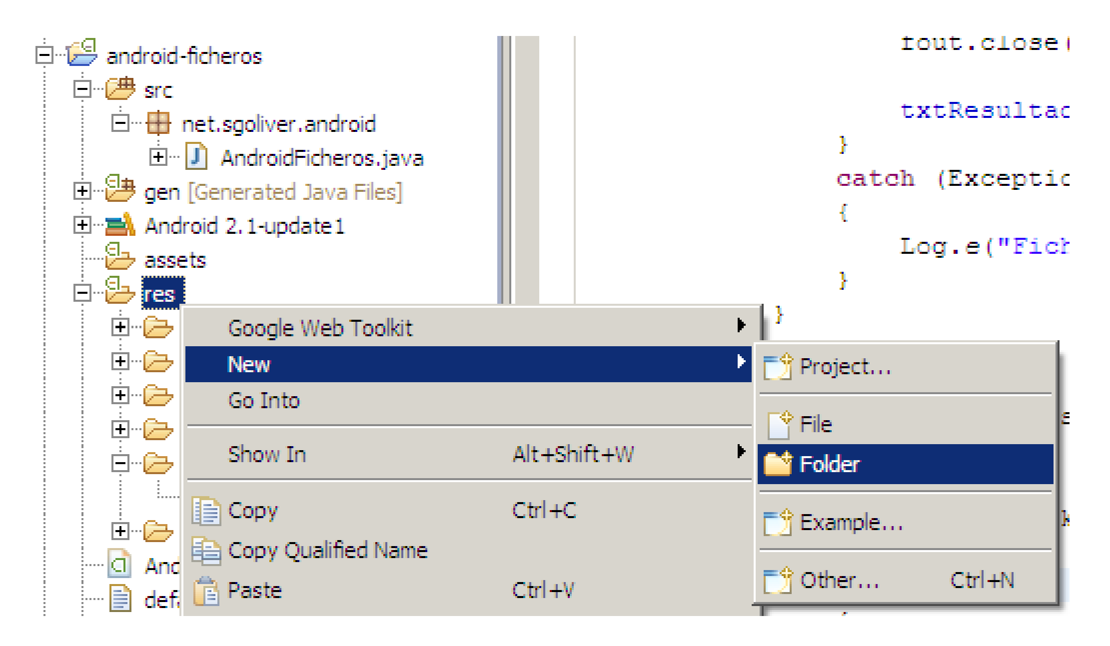
Por otra parte, leer ficheros desde la memoria interna es igual de sencillo, y procederemos de forma análoga, con la única diferencia de que utilizaremos el método `openFileInput()` para abrir el fichero, y los métodos de lectura de java.io para leer el contenido.

```
try
{
    BufferedReader fin =
        new BufferedReader(
            new InputStreamReader(
                openFileInput("prueba_int.txt")));

    String texto = fin.readLine();
    fin.close();
}
catch (Exception ex)
{
    Log.e("Ficheros", "Error al leer fichero desde memoria interna");
}
```

La segunda forma de almacenar ficheros en la memoria interna del dispositivo es incluirlos como recurso en la propia aplicación. Aunque este método es útil en muchos casos, sólo debemos utilizarlo cuando no necesitemos realizar modificaciones sobre los ficheros, ya que tendremos limitado el acceso a sólo lectura.

Para incluir un fichero como recurso de la aplicación debemos colocarlo en la carpeta "`/res/raw`" de nuestro proyecto de Eclipse. Esta carpeta no suele estar creada por defecto, por lo que deberemos crearla manualmente en Eclipse desde el menú contextual con la opción "*New / Folder*".



Una vez creada la carpeta /raw podremos colocar en ella cualquier fichero que queramos que se incluya con la aplicación en tiempo de compilación en forma de recurso. Nosotros incluiremos como ejemplo un fichero de texto llamado "prueba_raw.txt". Ya en tiempo de ejecución podremos acceder a este fichero, sólo en modo de lectura, de una forma similar a la que ya hemos visto para el resto de ficheros en memoria interna.

Para acceder al fichero, accederemos en primer lugar a los recursos de la aplicación con el método getResources() y sobre éstos utilizaremos el método openRawResource(id_del_recurso) para abrir el fichero en modo lectura. Este método devuelve un objeto InputStream, que ya podremos manipular como queramos mediante los métodos de la API java.io. Como ejemplo, nosotros convertiremos el stream en un objeto BufferedReader para leer el texto contenido en el fichero de ejemplo (por supuesto los ficheros de recurso también pueden ser binarios, como por ejemplo ficheros de imagen, video, etc). Veamos cómo quedaría el código:

```
try
{
    InputStream fraw =
        getResources().openRawResource(R.raw.prueba_raw);

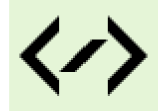
    BufferedReader brin =
        new BufferedReader(new InputStreamReader(fraw));

    String linea = brin.readLine();

    fraw.close();
}
catch (Exception ex)
{
    Log.e("Ficheros", "Error al leer fichero desde recurso raw");
}
```

Como puede verse en el código anterior, al método openRawResource() le pasamos como parámetro el ID del fichero incluido como recurso, que seguirá el patrón "R.raw.nombre_del_fichero", por lo que en nuestro caso particular será R.raw.prueba_raw.

La aplicación de ejemplo de este apartado consiste en una pantalla sencilla con 3 botones que realizan exactamente las tres tareas que hemos comentado: escribir y leer un fichero en la memoria interna, y leer un fichero de la carpeta raw.



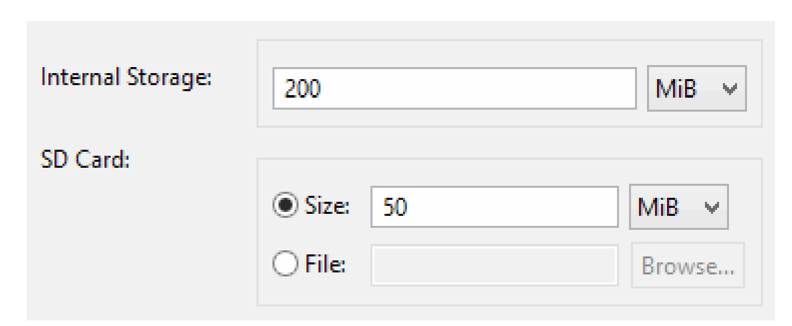
Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-ficheros-1

Ficheros en Memoria Externa (Tarjeta SD)

En el apartado anterior del curso hemos visto cómo manipular ficheros localizados en la memoria interna de un dispositivo Android. Sin embargo, como ya indicamos, esta memoria suele ser relativamente limitada y no es aconsejable almacenar en ella ficheros de gran tamaño. La alternativa natural es utilizar para ello la memoria externa del dispositivo, constituida normalmente por una tarjeta de memoria SD, aunque en dispositivos recientes también está presente en forma de almacenamiento no extraíble del dispositivo, aunque no por ello debe confundirse con la memoria interna. A diferencia de la memoria interna, el almacenamiento externo es público, es decir, todo lo que escribamos en él podrá ser leído por otras aplicaciones y por el usuario, por tanto hay que tener cierto cuidado a la hora de dicidir lo que escribimos en memoria interna y externa.

Una nota rápida antes de empezar con este tema. Para poder probar aplicaciones que hagan uso de la memoria externa en el emulador de Android necesitamos tener configurado en Eclipse un AVD que tenga establecido correctamente el tamaño de la tarjeta SD. En mi caso, he definido por ejemplo un tamaño de tarjeta de 50 Mb:



Seguimos. A diferencia de la memoria interna, la tarjeta de memoria no tiene por qué estar presente en el dispositivo, e incluso estándolo puede no estar reconocida por el sistema. Por tanto, el primer paso recomendado a la hora de trabajar con ficheros en memoria externa es asegurarnos de que dicha memoria está presente y disponible para leer y/o escribir en ella.

Para esto la API de Android proporciona (como método estático dela clase `Environment`) el método `getExternalStorageStatus()`, que no dice si la memoria externa está disponible y si se puede leer y escribir en ella. Este método devuelve una serie de valores que nos indicarán el estado de la memoria externa, siendo los más importantes los siguientes:

- `MEDIA_MOUNTED`, que indica que la memoria externa está disponible y podemos tanto leer como escribir en ella.
- `MEDIA_MOUNTED_READ_ONLY`, que indica que la memoria externa está disponible pero sólo podemos leer de ella.
- Otra serie de valores que indicarán que existe algún problema y que por tanto no podemos ni leer ni escribir en la memoria externa (`MEDIA_UNMOUNTED`, `MEDIA_REMOVED`, ...). Podéis consultar todos estos estados en la documentación oficial de la clase `Environment`.

Con todo esto en cuenta, podríamos realizar un chequeo previo del estado de la memoria externa del dispositivo de la siguiente forma:

```
boolean sdDisponible = false;
boolean sdAccesoEscritura = false;

//Comprobamos el estado de la memoria externa (tarjeta SD)
String estado = Environment.getExternalStorageState();

if (estado.equals(Environment.MEDIA_MOUNTED))
{
    sdDisponible = true;
    sdAccesoEscritura = true;
}
else if (estado.equals(Environment.MEDIA_MOUNTED_READ_ONLY))
{
    sdDisponible = true;
    sdAccesoEscritura = false;
}
else
{
    sdDisponible = false;
    sdAccesoEscritura = false;
}
```

Una vez chequeado el estado de la memoria externa, y dependiendo del resultado obtenido, ya podremos leer o escribir en ella cualquier tipo de fichero.

Empecemos por la escritura. Para escribir un fichero a la memoria externa tenemos que obtener en primer lugar la ruta al directorio raíz de esta memoria. Para ello podemos utilizar el método `getExternalStorageDirectory()` de la clase `Environment`, que nos devolverá un objeto `File` con la ruta de dicho directorio. A partir de este objeto, podremos construir otro con el nombre elegido para nuestro fichero (como ejemplo "`prueba_sd.txt`"), creando un nuevo objeto `File` que combine ambos elementos. Tras esto, ya sólo queda encapsularlo en algún objeto de escritura de ficheros de la API de java y escribir algún dato de prueba. En nuestro caso de ejemplo lo convertiremos una vez más a un objeto `OutputStreamWriter` para escribir al fichero un mensaje de texto. Veamos cómo quedaría el código:

```
try
{
    File ruta_sd = Environment.getExternalStorageDirectory();

    File f = new File(ruta_sd.getAbsolutePath(), "prueba_sd.txt");

    OutputStreamWriter fout =
        new OutputStreamWriter(
            new FileOutputStream(f));

    fout.write("Texto de prueba.");
    fout.close();
} catch (Exception ex)
{
    Log.e("Ficheros", "Error al escribir fichero a tarjeta SD");
}
```

El código anterior funcioa sin problemas pero escribirá el fichero directamente en la carpeta raíz de la memoria externa. Esto, aunque en ocasiones puede resultar necesario, no es una buena práctica. Lo correcto sería disponer de una carpeta propia para nuestra aplicación, lo que además tendrá la ventaja de que al desinstalar la aplicación también se liberará este espacio. Esto lo conseguimos utilizando el método `getExternalFilesDir(null)` en vez de `getExternalStorageDirectory()`. El método `getExternalFilesDir()` nos devuelve directamente la ruta de una carpeta específica para nuestra

aplicación dentro de la memoria externa siguiendo el siguiente patrón:

<raíz_mem_ext>/Android/data/nuestro.paquete.java/files

Si en vez de null le indicamos como parámetro un tipo de datos determinado (DIRECTORY_MUSIC, DIRECTORY_PICTURES, DIRECTORY_MOVIES, DIRECTORY_RINGTONES, DIRECTORY_ALARMS, DIRECTORY_NOTIFICATIONS, DIRECTORY_PODCASTS) nos devolverá una subcarpeta dentro de la anterior con su nombre correspondiente. Así, por ejemplo, una llamada al método getExternalFilesDir(Environment.DIRECTORY_MUSIC) nos devolvería la siguiente carpeta:

<raíz_mem_ext>/Android/data/nuestro.paquete.java/files/Music

Esto último, además, ayuda a Android a saber qué tipo de contenidos hay en cada carpeta, de forma que puedan clasificarse correctamente por ejemplo en la galería multimedia.

Sea como sea, para tener acceso a la memoria externa tendremos que especificar en el fichero AndroidManifest.xml que nuestra aplicación necesita permiso de escritura en dicha memoria. Para añadir un nuevo permiso usaremos como siempre la cláusula <uses-permission> utilizando el valor concreto "android.permission.WRITE_EXTERNAL_STORAGE". Con esto, nuestro AndroidManifest.xml quedaría de forma similar a éste:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="net.sgoliver.android.ficheros"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

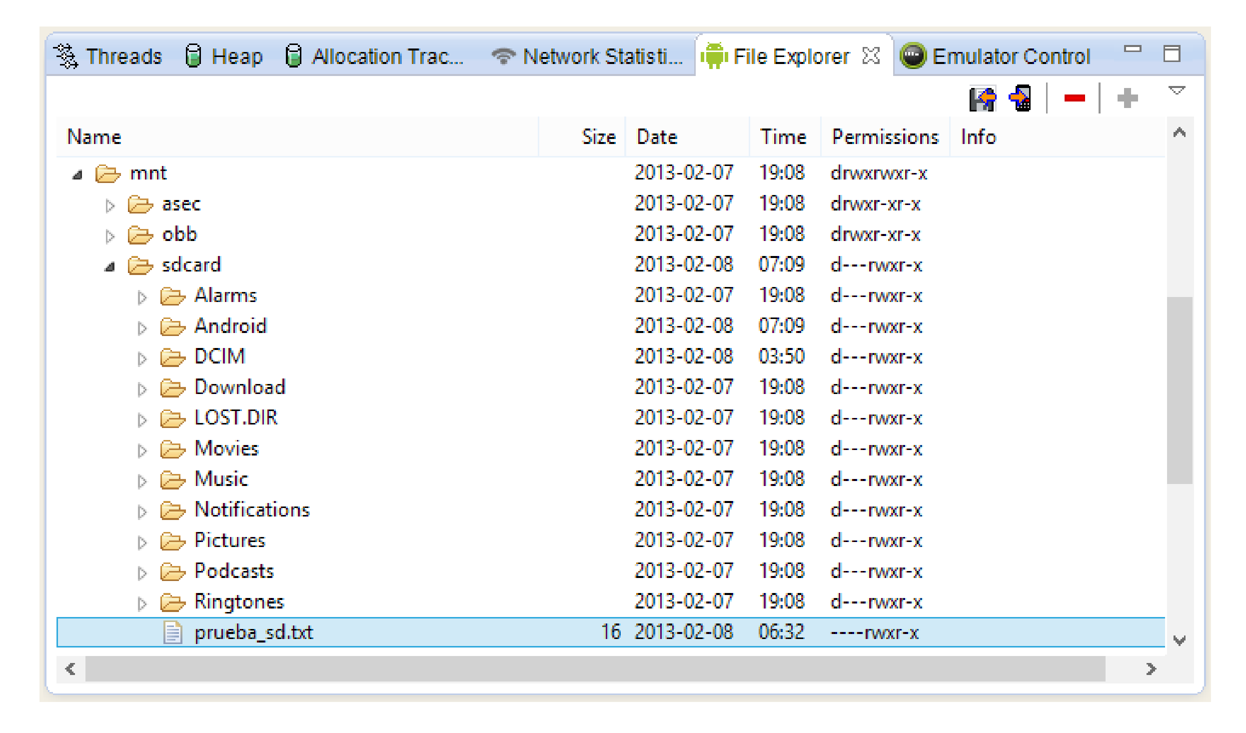
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE">
    </uses-permission>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="net.sgoliver.android.ficheros.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Si ejecutamos ahora el código y nos vamos al explorador de archivos del DDMS podremos comprobar cómo se ha creado correctamente el fichero en el directorio raíz de nuestra SD. Esta ruta puede variar entre dispositivos, pero para Android 2.x suele localizarse en la carpeta /sdcard/, mientras que para Android 4 suele estar en /mnt/sdcard/.



Por su parte, leer un fichero desde la memoria externa es igual de sencillo. Obtenemos el directorio raíz de la memoria externa con `getExternalStorageDirectory()` o la carpeta específica de nuestra aplicación con `getExternalFilesDir()` como ya hemos visto, creamos un objeto `File` que combine esa ruta con el nombre del fichero a leer y lo encapsulamos dentro de algún objeto que facilite la lectura, nosotros para leer texto utilizaremos como siempre un `BufferedReader`.

```
try
{
    File ruta_sd = Environment.getExternalStorageDirectory();

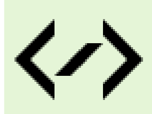
    File f = new File(ruta_sd.getAbsolutePath(), "prueba_sd.txt");

    BufferedReader fin =
        new BufferedReader(
            new InputStreamReader(
                new FileInputStream(f)));

    String texto = fin.readLine();
    fin.close();
}
catch (Exception ex)
{
    Log.e("Ficheros", "Error al leer fichero desde tarjeta SD");
}
```

Como vemos, el código es análogo al que hemos visto para la escritura de ficheros.

Como aplicación de ejemplo de este apartado he partido de la desarrollada en el anterior dedicado a la memoria interna y he añadido dos nuevos botones para leer y escribir a memoria externa tal como hemos descrito. Los resultados se muestran en el log de la aplicación.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-ficheros-2

11

Content Providers

XI. Content Providers

Construcción de Content Providers

En este nuevo apartado del curso vamos a tratar el temido [o a veces incomprendido] tema de los *Content Providers*.

Un *Content Provider* no es más que el mecanismo proporcionado por la plataforma Android para permitir compartir información entre aplicaciones. Una aplicación que desee que todo o parte de la información que almacena esté disponible de una forma controlada para el resto de aplicaciones del sistema deberá proporcionar un *content provider* a través del cual se pueda realizar el acceso a dicha información. Este mecanismo es utilizado por muchas de las aplicaciones estándar de un dispositivo Android, como por ejemplo la lista de contactos, la aplicación de SMS, o el calendario/agenda. Esto quiere decir que podríamos acceder a los datos gestionados por estas aplicaciones desde nuestras propias aplicaciones Android haciendo uso de los *content providers* correspondientes.

Son por tanto dos temas los que debemos tratar en este apartado, por un lado a construir nuevos *content providers* personalizados para nuestras aplicaciones, y por otro utilizar un *content provider* ya existente para acceder a los datos publicados por otras aplicaciones.

En gran parte de la bibliografía sobre programación en Android se suele tratar primero el tema del acceso a *content providers* ya existentes (como por ejemplo el acceso a la lista de contactos de Android) para después pasar a la construcción de nuevos *content providers* personalizados. Yo sin embargo voy a tratar de hacerlo en orden inverso, ya que me parece importante conocer un poco el funcionamiento interno de un *content provider* antes de pasar a utilizarlo sin más dentro de nuestras aplicaciones. Así, en este primer apartado sobre el tema veremos cómo crear nuestro propio *content provider* para compartir datos con otras aplicaciones, y en el próximo apartado veremos cómo utilizar este mecanismo para acceder directamente a datos de terceros.

Empecemos a entrar en materia. Para añadir un *content provider* a nuestra aplicación tendremos que:

1. Crear una nueva clase que extienda a la clase android `ContentProvider`.
2. Declarar el nuevo *content provider* en nuestro fichero AndroidManifest.xml

Por supuesto nuestra aplicación tendrá que contar previamente con algún método de almacenamiento interno para la información que queremos compartir. Lo más común será disponer de una base de datos SQLite, por lo que será esto lo que utilizaré para todos los ejemplos de este apartado, pero internamente podríamos tener los datos almacenados de cualquier otra forma, por ejemplo en ficheros de texto, ficheros XML, etc. El content provider será el mecanismo que nos permita publicar esos datos a terceros de una forma homogénea y a través de una interfaz estandarizada.

Un primer detalle a tener en cuenta es que los registros de datos proporcionados por un content provider deben contar siempre con un campo llamado `_ID` que los identifique de forma unívoca del resto de registros. Como ejemplo, los registros devueltos por un content provider de clientes podría tener este aspecto:

_ID	Cliente	Telefono	Email
3	Antonio	900123456	email1@correo.com
7	Jose	900123123	email2@correo.com
9	Luis	900123987	email3@correo.com

Sabiendo esto, es interesante que nuestros datos también cuenten internamente con este campo `_ID` (no tiene por qué llamarse igual) de forma que nos sea más sencillo después generar los resultados del content provider.

Con todo esto, y para tener algo desde lo que partir, vamos a construir en primer lugar una aplicación de ejemplo muy sencilla con una base de datos SQLite que almacene los datos de una serie de clientes con una estructura similar a la tabla anterior. Para ello seguiremos los mismos pasos que ya comentamos en los apartados dedicados al tratamiento de bases de datos SQLite en Android (consultar índice del curso).

Por volver a recordarlo muy brevemente, lo que haremos será crear una nueva clase que extienda a `SQLiteOpenHelper`, definiremos las sentencias SQL para crear nuestra tabla de clientes, e implementaremos finalmente los métodos `onCreate()` y `onUpgrade()`. El código de esta nueva clase, que yo he llamado `ClientesSqliteHelper`, quedaría como sigue:

```
public class ClientesSqliteHelper extends SQLiteOpenHelper {

    //Sentencia SQL para crear la tabla de Clientes
    String sqlCreate = "CREATE TABLE Clientes " +
                       "(_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                       " nombre TEXT, " +
                       " telefono TEXT, " +
                       " email TEXT )";

    public ClientesSqliteHelper(Context contexto, String nombre,
                                CursorFactory factory, int version) {

        super(contexto, nombre, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        //Se ejecuta la sentencia SQL de creación de la tabla
        db.execSQL(sqlCreate);

        //Insertamos 15 clientes de ejemplo
        for(int i=1; i<=15; i++)
        {
            //Generamos los datos de muestra
            String nombre = "Cliente" + i;
            String telefono = "900-123-00" + i;
            String email = "email" + i + "@mail.com";

            //Insertamos los datos en la tabla Clientes
            db.execSQL("INSERT INTO Clientes (nombre, telefono, email) " +
                       "VALUES ('" + nombre + "', '" + telefono +"', '" +
                       email + "')");
        }
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int versionAnterior,
                         int versionNueva) {

        //NOTA: Por simplicidad del ejemplo aquí utilizamos directamente la
        //      opción de eliminar la tabla anterior y crearla de nuevo
        //      vacía con el nuevo formato.
        //      Sin embargo lo normal será que haya que migrar datos de la
        //      tabla antigua a la nueva, por lo que este método debería
        //      ser más elaborado.
```

```
            //Se elimina la versión anterior de la tabla
            db.execSQL("DROP TABLE IF EXISTS Clientes");

            //Se crea la nueva versión de la tabla
            db.execSQL(sqlCreate);
        }
}
```

Como notas relevantes del código anterior:

- Nótese el campo "`_id`" que hemos incluido en la base de datos de clientes por lo motivos indicados un poco más arriba. Este campo lo declaramos como `INTEGER PRIMARY KEY AUTOINCREMENT`, de forma que se incremente automáticamente cada vez que insertamos un nuevo registro en la base de datos.

- En el método `onCreate()`, además de ejecutar la sentencia SQL para crear la tabla Clientes, también inserta varios registros de ejemplo.

- Para simplificar el ejemplo, el método `onUpgrade()` se limita a eliminar la tabla actual y crear una nueva con la nueva estructura. En una aplicación real habría que hacer probablemente la migración de los datos a la nueva base de datos.

Dado que la clase anterior ya se ocupa de todo, incluso de insertar algunos registro de ejemplo con los que podamos hacer pruebas, la aplicación principal de ejemplo no mostrará en principio nada en pantalla ni hará nada con la información. Esto lo he decidido así para no complicar el código de la aplicación innecesariamente, ya que no nos va a interesar el tratamiento directo de los datos por parte de la aplicación principal, sino su utilización a través del content provider que vamos a construir.

Una vez que ya contamos con nuestra aplicación de ejemplo y su base de datos, es hora de empezar a construir el nuevo content provider que nos permitirá compartir sus datos con otras aplicaciones.

Lo primero que vamos a comentar es la forma con que se hace referencia en Android a los content providers. El acceso a un content provider se realiza siempre mediante una URI. Una URI no es más que una cadena de texto similar a cualquiera de las direcciones web que utilizamos en nuestro navegador. Al igual que para acceder a mi blog lo hacemos mediante la dirección "http://www.sgoliver.net", para acceder a un content provider utilizaremos una dirección similar a:

"`content://net.sgoliver.android.contentproviders/clientes`".

Las direcciones URI de los content providers están formadas por 3 partes. En primer lugar el prefijo "`content://`" que indica que dicho recurso deberá ser tratado por un content provider. En segundo lugar se indica el identificador en sí del content provider, también llamado *authority*. Dado que este dato debe ser único es una buena práctica utilizar un authority de tipo "nombre de clase java invertido", por ejemplo en mi caso "`net.sgoliver.android.contentproviders`". Por último, se indica la entidad concreta a la que queremos acceder dentro de los datos que proporciona el content provider. En nuestro caso será simplemente la tabla de "`clientes`", ya que será la única existente, pero dado que un content provider puede contener los datos de varias entidades distintas en este último tramo de la URI habrá que especificarlo. Indicar por último que en una URI se puede hacer referencia directamente a un registro concreto de la entidad seleccionada. Esto se haría indicando al final de la URI el ID de dicho registro. Por ejemplo la uri "`content://net.sgoliver.android.contentproviders/clientes/23`" haría referencia directa al cliente con _ID = 23.

Todo esto es importante ya que será nuestro content provider el encargado de interpretar/parsear la URI completa para determinar los datos que se le están solicitando. Esto lo veremos un poco más adelante.

Sigamos. El siguiente paso será extender a la clase `ContentProvider`.

Si echamos un vistazo a los métodos abstractos que tendremos que implementar veremos que tenemos los

siguientes:

- `onCreate()`
- `query()`
- `insert()`
- `update()`
- `delete()`
- `getType()`

El primero de ellos nos servirá para inicializar todos los recursos necesarios para el funcionamiento del nuevo content provider. Los cuatro siguientes serán los métodos que permitirán acceder a los datos (consulta, inserción, modificación y eliminación, respectivamente) y por último, el método `getType()` permitirá conocer el tipo de datos devueltos por el content provider (más tarde intentaremos explicar algo mejor esto último).

Además de implementar estos métodos, también definiremos una serie de constantes dentro de nuestra nueva clase provider, que ayudarán posteriormente a su utilización. Veamos esto paso a paso. Vamos a crear una nueva clase `ClientesProvider` que extienda de `ContentProvider`.

Lo primero que vamos a definir es la URI con la que se accederá a nuestro content provider. En mi caso he elegido la siguiente:

`content://net.sgoliver.android.contentproviders/clientes`

Además, para seguir la práctica habitual de todos los content providers de Android, encapsularemos además esta dirección en un objeto estático de tipo Uri llamado `CONTENT_URI`.

```
//Definición del CONTENT_URI
private static final String uri =
    "content://net.sgoliver.android.contentproviders/clientes";

public static final Uri CONTENT_URI = Uri.parse(uri);
```

A continuación vamos a definir varias constantes con los nombres de las columnas de los datos proporcionados por nuestro content provider. Como ya dijimos antes existen columnas predefinidas que deben tener todos los content providers, por ejemplo la columna _ID. Estas columnas estándar están definidas en la clase `BaseColumns`, por lo que para añadir las nuevas columnas de nuestro content provider definiremos una clase interna pública tomando como base la clase `BaseColumns` y añadiremos nuestras nuevas columnas.

```
//Clase interna para declarar las constantes de columna
public static final class Clientes implements BaseColumns
{
    private Clientes() {}

    //Nombres de columnas
    public static final String COL_NOMBRE = "nombre";
    public static final String COL_TELEFONO = "telefono";
    public static final String COL_EMAIL = "email";
}
```

Por último, vamos a definir varios atributos privados auxiliares para almacenar el nombre de la base de datos, la versión, y la tabla a la que accederá nuestro content provider.

```
//Base de datos
private ClientesSqliteHelper clidbh;
private static final String BD_NOMBRE = "DBClientes";
private static final int BD_VERSION = 1;
private static final String TABLA_CLIENTES = "Clientes";
```

Como se indicó anteriormente, la primera tarea que nuestro content provider deberá hacer cuando se acceda a él será interpretar la URI utilizada. Para facilitar esta tarea Android proporciona una clase llamada `UriMatcher`, capaz de interpretar determinados patrones en una URI. Esto nos será útil para determinar por ejemplo si una URI hace referencia a una tabla genérica o a un registro concreto a través de su ID. Por ejemplo:

- `content://net.sgoliver.android.contentproviders/clientes` –> Acceso genérico a tabla de clientes
- `content://net.sgoliver.android.contentproviders/clientes/17` –> Acceso directo al cliente con ID = 17

Para conseguir esto definiremos también como miembro de la clase un objeto `UriMatcher` y dos nuevas constantes que representen los dos tipos de URI que hemos indicado: acceso genérico a tabla (lo llamaré `CLIENTES`) o acceso a cliente por ID (lo llamaré `CLIENTES_ID`). A continuación inicializaremos el objeto `UriMatcher` indicándole el formato de ambos tipos de URI, de forma que pueda diferenciarlos y devolvernos su tipo (una de las dos constantes definidas, `CLIENTES` o `CLIENTES_ID`).

```
//UriMatcher
private static final int CLIENTES = 1;
private static final int CLIENTES_ID = 2;
private static final UriMatcher uriMatcher;

//Inicializamos el UriMatcher
static {
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);

    uriMatcher.addURI("net.sgoliver.android.contentproviders",
                      "clientes", CLIENTES);

    uriMatcher.addURI("net.sgoliver.android.contentproviders",
                      "clientes/#", CLIENTES_ID);
}
```

En el código anterior vemos como mediante el método `addUri()` indicamos el *authority* de nuestra URI, el formato de la entidad que estamos solicitando, y el tipo con el que queremos identificar dicho formato. Más tarde veremos cómo utilizar esto de forma práctica.

Bien, pues ya tenemos definidos todos los miembros necesarios para nuestro nuevo content provider. Ahora toca implementar los métodos comentados anteriormente.

El primero de ellos es `onCreate()`. En este método nos limitaremos simplemente a inicializar nuestra base de datos, a través de su nombre y versión, y utilizando para ello la clase `ClientesSqliteHelper` que creamos al principio del apartado.

```
@Override
public boolean onCreate() {

    clidbh = new ClientesSqliteHelper(
            getContext(), BD_NOMBRE, null, BD_VERSION);

    return true;
}
```

La parte interesante llega con el método query(). Este método recibe como parámetros una URI, una lista de nombres de columna, un criterio de selección, una lista de valores para las variables utilizadas en el criterio anterior, y un criterio de ordenación. Todos estos datos son análogos a los que comentamos cuando tratamos la consulta de datos en SQLite para Android, apartado que recomiendo releer si no tenéis muy frescos estos conocimientos. El método query deberá devolver los datos solicitados según la URI indicada y los criterios de selección y ordenación pasados como parámetro. Así, si la URI hace referencia a un cliente concreto por su ID ése deberá ser el único registro devuelto. Si por el contrario es un acceso genérico a la tabla de clientes habrá que realizar la consulta SQL correspondiente a la base de datos respetando los criterios pasados como parámetro.

Para distinguir entre los dos tipos de URI posibles utilizaremos como ya hemos indicado el objeto uriMatcher, utilizando su método match(). Si el tipo devuelto es CLIENTES_ID, es decir, que se trata de un acceso a un cliente concreto, sustituiremos el criterio de selección por uno que acceda a la tabla de clientes sólo por el ID indicado en la URI. Para obtener este ID utilizaremos el método getLastPathSegment() del objeto uri que extrae el último elemento de la URI, en este caso el ID del cliente.

Hecho esto, ya tan sólo queda realizar la consulta a la base de datos mediante el método query() de SQLiteDatabase. Esto es sencillo ya que los parámetros son análogos a los recibidos en el método query() del content provider.

```
@Override
public Cursor query(Uri uri, String[] projection,
    String selection, String[] selectionArgs, String sortOrder) {

    //Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;
    if(uriMatcher.match(uri) == CLIENTES_ID){
                where = "_id=" + uri.getLastPathSegment();
        }

    SQLiteDatabase db = clidbh.getWritableDatabase();

    Cursor c = db.query(TABLA_CLIENTES, projection, where,
                    selectionArgs, null, null, sortOrder);

    return c;
}
```

Como vemos, los resultados se devuelven en forma de objeto Cursor, una vez más exactamente igual a como lo hace el método query() de SQLiteDatabase.

Por su parte, los métodos update() y delete() son completamente análogos a éste, con la única diferencia de que devuelven el número de registros afectados en vez de un cursor a los resultados. Vemos directamente el código:

```
@Override
public int update(Uri uri, ContentValues values,
              String selection, String[] selectionArgs) {

    int cont;

    //Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;
    if(uriMatcher.match(uri) == CLIENTES_ID){
            where = "_id=" + uri.getLastPathSegment();
    }

    SQLiteDatabase db = clidbh.getWritableDatabase();

    cont = db.update(TABLA_CLIENTES, values, where, selectionArgs);

    return cont;
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {

    int cont;

    //Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;

    if(uriMatcher.match(uri) == CLIENTES_ID){
         where = "_id=" + uri.getLastPathSegment();
    }

    SQLiteDatabase db = clidbh.getWritableDatabase();

    cont = db.delete(TABLA_CLIENTES, where, selectionArgs);

    return cont;
}
```

El método `insert()` sí es algo diferente, aunque igual de sencillo. La diferencia en este caso radica en que debe devolver la URI que hace referencia al nuevo registro insertado. Para ello, obtendremos el nuevo ID del elemento insertado como resultado del método `insert()` de `SQLiteDatabase`, y posteriormente construiremos la nueva URI mediante el método auxiliar `ContentUris.withAppendedId()` que recibe como parámetro la URI de nuestro content provider y el ID del nuevo elemento.

```
@Override
public Uri insert(Uri uri, ContentValues values) {

    long regId = 1;

    SQLiteDatabase db = clidbh.getWritableDatabase();

    regId = db.insert(TABLA_CLIENTES, null, values);

    Uri newUri = ContentUris.withAppendedId(CONTENT_URI, regId);

    return newUri;
}
```

Por último, tan sólo nos queda implementar el método `getType()`. Este método se utiliza para identificar el tipo de datos que devuelve el content provider. Este tipo de datos se expresará como un *MIME Type*, al igual que hacen los navegadores web para determinar el tipo de datos que están recibiendo tras una petición a un servidor. Identificar el tipo de datos que devuelve un content provider ayudará por ejemplo a Android a determinar qué aplicaciones son capaces de procesar dichos datos.

Una vez más existirán dos tipos MIME distintos para cada entidad del content provider, uno de ellos destinado a cuando se devuelve una lista de registros como resultado, y otro para cuando se devuelve un registro único concreto. De esta forma, seguiremos los siguientes patrones para definir uno y otro tipo de datos:

- "`vnd.android.cursor.item/vnd.xxxxxx`" –> Registro único
- "`vnd.android.cursor.dir/vnd.xxxxxx`" –> Lista de registros

En mi caso de ejemplo, he definido los siguientes tipos:

- "`vnd.android.cursor.item/vnd.sgoliver.cliente`"
- "`vnd.android.cursor.dir/vnd.sgoliver.cliente`"

Con esto en cuenta, la implementación del método `getType()` quedaría como sigue:

```
@Override
public String getType(Uri uri) {

    int match = uriMatcher.match(uri);

    switch (match)
    {
        case CLIENTES:
            return "vnd.android.cursor.dir/vnd.sgoliver.cliente";
        case CLIENTES_ID:
            return "vnd.android.cursor.item/vnd.sgoliver.cliente";
        default:
            return null;
    }
}
```

Como se puede observar, utilizamos una vez más el objeto `UriMatcher` para determinar el tipo de URI que se está solicitando y en función de ésta devolvemos un tipo MIME u otro.

Pues bien, con esto ya hemos completado la implementación del nuevo content provider. Pero aún nos queda un paso más, como indicamos al principio del apartado. Debemos declarar el content provider en nuestro fichero `AndroidManifest.xml` de forma que una vez instalada la aplicación en el dispositivo Android conozca la existencia de dicho recurso. Para ello, bastará insertar un nuevo elemento `<provider>` dentro de `<application>` indicando el nombre del content provider y su *authority*.

```
<application android:icon="@drawable/icon"
        android:label="@string/app_name">

        ...

        <provider android:name="ClientesProvider"
        android:authorities="net.sgoliver.android.contentproviders"/>

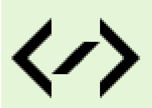
</application>
```

Ahora sí hemos completado totalmente la construcción de nuestro nuevo content provider mediante el cual

otras aplicaciones del sistema podrán acceder a los datos almacenados por nuestra aplicación.

En el siguiente apartado veremos cómo utilizar este nuevo content provider para acceder a los datos de nuestra aplicación de ejemplo, y también veremos cómo podemos utilizar alguno de los content provider predefinidos por Android para consultar datos del sistema, como por ejemplo la lista de contactos o la lista de llamadas realizadas.

El código fuente completo de la aplicación lo publicaré junto al siguiente apartado, pero por el momento podéis descargar desde este enlace los fuentes de las dos clases implementadas en este apartado y el fichero `AndroidManifest.xml` modificado.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-content-providers-1

Utilización de Content Providers

En el apartado anterior vimos cómo construir un *content provider* personalizado para permitir a nuestras aplicaciones Android compartir datos con otras aplicaciones del sistema. En este nuevo apartado vamos a ver el tema desde el punto de vista opuesto, es decir, vamos a aprender a hacer uso de un content provider ya existente para acceder a datos de otras aplicaciones. Además, veremos cómo también podemos acceder a datos del propio sistema Android (logs de llamadas, lista de contactos, agenda telefónica, bandeja de entrada de sms, etc) utilizando este mismo mecanismo.

Vamos a comenzar explicando cómo podemos utilizar el content provider que implementamos en el apartado anterior para acceder a los datos de los clientes. Para no complicar mucho el ejemplo ni hacer más difícil las pruebas y la depuración en el emulador de Android vamos a hacer uso el content provider desde la propia aplicación de ejemplo que hemos creado. De cualquier forma, el código necesario sería exactamente igual si lo hiciéramos desde otra aplicación distinta.

Utilizar un content provider ya existente es muy sencillo, sobre todo comparado con el laborioso proceso de construcción de uno nuevo. Para comenzar, debemos obtener una referencia a un *Content Resolver*, objeto a través del que realizaremos todas las acciones necesarias sobre el content provider. Esto es tan fácil como utilizar el método `getContentResolver()` desde nuestra actividad para obtener la referencia indicada. Una vez obtenida la referencia al content resolver, podremos utilizar sus métodos `query()`, `update()`, `insert()` y `delete()` para realizar las acciones equivalentes sobre el content provider. Por ver varios ejemplos de la utilización de estos métodos añadiremos a nuestra aplicación de ejemplo tres botones en la pantalla principal, uno para hacer una consulta de todos los clientes, otro para insertar registros nuevos, y el último para eliminar todos los registros nuevos insertados con el segundo botón.

Empecemos por la consulta de clientes. El procedimiento será prácticamente igual al que vimos en los apartados de acceso a bases de datos SQLite (consultar el índice del curso). Comenzaremos por definir un *array* con los nombres de las columnas de la tabla que queremos recuperar en los resultados de la consulta, que en nuestro caso serán el ID, el nombre, el teléfono y el email. Tras esto, obtendremos como dijimos antes una referencia al content resolver y utilizaremos su método `query()` para obtener los resultados en forma de cursor. El método `query()` recibe, como ya vimos en el apartado anterior, la Uri del content provider al que queremos acceder, el array de columnas que queremos recuperar, el criterio de selección, los argumentos variables, y el criterio de ordenación de los resultados. En nuestro caso, para no complicarnos utilizaremos tan sólo los dos primeros, pasándole el `CONTENT_URI` de nuestro provider y el array de columnas que acabamos de definir.

```
//Columnas de la tabla a recuperar
String[] projection = new String[] {
    Clientes._ID,
    Clientes.COL_NOMBRE,
    Clientes.COL_TELEFONO,
    Clientes.COL_EMAIL };

Uri clientesUri =  ClientesProvider.CONTENT_URI;

ContentResolver cr = getContentResolver();

//Hacemos la consulta
Cursor cur = cr.query(clientesUri,
        projection, //Columnas a devolver
        null,       //Condición de la query
        null,       //Argumentos variables de la query
        null);      //Orden de los resultados
```

Hecho esto, tendremos que recorrer el cursor para procesar los resultados. Para nuestro ejemplo, simplemente los escribiremos en un cuadro de texto (`txtResultados`) colocado bajo los tres botones de ejemplo. Una vez más, si tienes dudas sobre cómo recorrer un cursor, puedes consultar los apartados del curso dedicados al tratamiento de bases de datos SQLite, por ejemplo éste. Veamos cómo quedaría el código:

```
if (cur.moveToFirst())
{
    String nombre;
    String telefono;
    String email;

    int colNombre = cur.getColumnIndex(Clientes.COL_NOMBRE);
    int colTelefono = cur.getColumnIndex(Clientes.COL_TELEFONO);
    int colEmail = cur.getColumnIndex(Clientes.COL_EMAIL);

    txtResultados.setText("");

    do
    {
        nombre = cur.getString(colNombre);
        telefono = cur.getString(colTelefono);
        email = cur.getString(colEmail);

        txtResultados.append(nombre + " - " + telefono + " - " + email +
"\n");

    } while (cur.moveToNext());
}
```

Para insertar nuevos registros, el trabajo será también exactamente igual al que se hace al tratar directamente con bases de datos SQLite. Rellenaremos en primer lugar un objeto `ContentValues` con los datos del nuevo cliente y posteriormente utilizamos el método `insert()` pasándole la URI del content provider en primer lugar, y los datos del nuevo registro como segundo parámetro.

```
ContentValues values = new ContentValues();

values.put(Clientes.COL_NOMBRE, "ClienteN");
values.put(Clientes.COL_TELEFONO, "999111222");
values.put(Clientes.COL_EMAIL, "nuevo@email.com");

ContentResolver cr = getContentResolver();

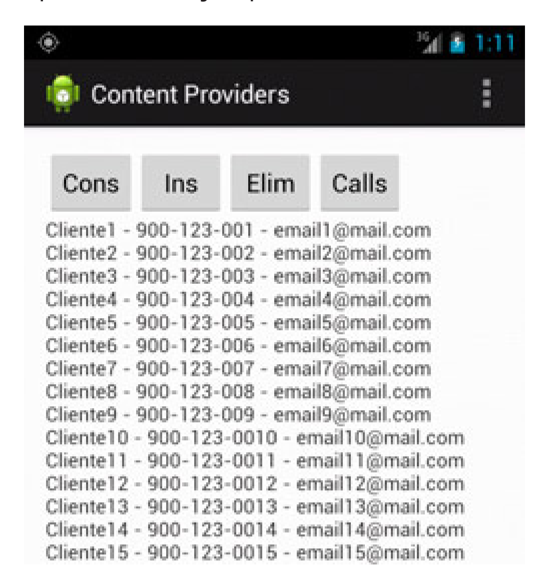
cr.insert(ClientesProvider.CONTENT_URI, values);
```

Por último, y más sencillo todavía, la eliminación de registros la haremos directamente utilizando el método `delete()` del content resolver, indicando como segundo parámetro el criterio de localización de los registros que queremos eliminar, que en este caso serán los que hayamos insertado nuevos con el segundo botón de ejemplo (aquellos con nombre = 'ClienteN').

```
ContentResolver cr = getContentResolver();

cr.delete(ClientesProvider.CONTENT_URI,
        Clientes.COL_NOMBRE + " = 'ClienteN'", null);
```

Como muestra gráfica, veamos por ejemplo el resultado de la consulta de clientes (primer botón) en la aplicación de ejemplo.

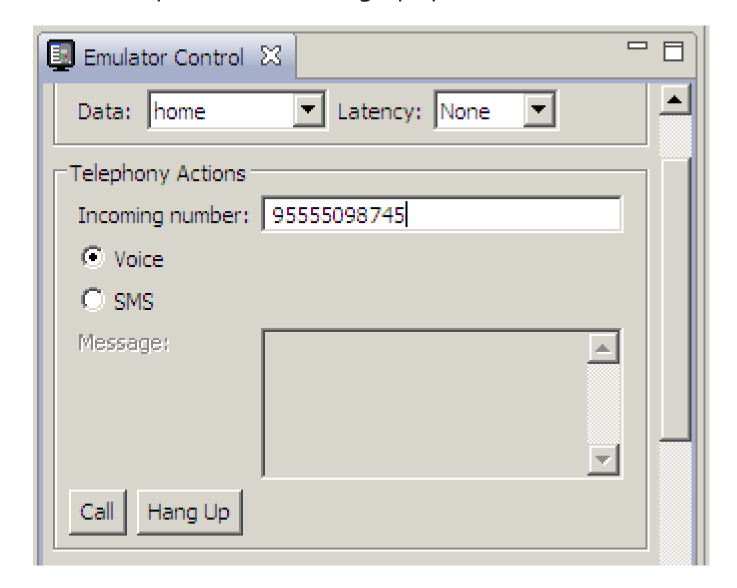


Con esto, hemos visto lo sencillo que resulta acceder a los datos proporcionados por un content provider. Pues bien, éste es el mismo mecanismo que podemos utilizar para acceder a muchos datos de la propia plataforma Android. En la documentación oficial del paquete android.provider podemos consultar los datos que tenemos disponibles a través de este mecanismo, entre ellos encontramos por ejemplo: el historial de llamadas, la agenda de contactos y teléfonos, las bibliotecas multimedia (audio y video), o el historial y la lista de favoritos del navegador.

Por ver un ejemplo de acceso a este tipo de datos, vamos a realizar una consulta al historial de llamadas del dispositivo, para lo que accederemos al content provider `android.provider.CallLog`.

En primer lugar vamos a registrar varias llamadas en el emulador de Android, de forma que los resultados de la consulta al historial de llamadas contenga algunos registros. Haremos por ejemplo varias llamadas salientes desde el emulador y simularemos varias llamadas entrantes desde el DDMS. Las primeras son sencillas, simplemente ve al emulador, accede al teléfono, marca y descuelga igual que lo harías en un dispositivo físico. Y para emular llamadas entrantes podremos hacerlo una vez más desde Eclipse, accediendo a la vista del DDMS. En esta vista, si accedemos a la sección "*Emulator Control*" veremos un apartado llamado "*Telephony Actions*". Desde éste, podemos introducir un número de teléfono origen cualquiera y pulsar el

botón "*Call*" para conseguir que nuestro emulador reciba una llamada entrante. Sin aceptar la llamada en el emulador pulsaremos "*Hang Up*" para terminar la llamada simulando así una llamada perdida.



Hecho esto, procedemos a realizar la consulta al historial de llamadas utilizando el content provider indicado, y para ello añadiremos un botón más a la aplicación de ejemplo.

Consultando la documentación del content provider veremos que podemos extraer diferentes datos relacionados con la lista de llamadas. Nosotros nos quedaremos sólo con dos significativos, el número origen o destino de la llamada, y el tipo de llamada (entrante, saliente, perdida). Los nombres de estas columnas se almacenan en las constantes `Calls.NUMBER` y `Calls.TYPE` respectivamente.

Decidido esto, actuaremos igual que antes. Definiremos el array con las columnas que queremos recuperar, obtendremos la referencia al content resolver y ejecutaremos la consulta llamando al método `query()`. Por último, recorremos el cursor obtenido y procesamos los resultados. Igual que antes, lo único que haremos será escribir los resultados al cuadro de texto situado bajo los botones. Veamos el código:

```
String[] projection = new String[] {
    Calls.TYPE,
    Calls.NUMBER };

Uri llamadasUri =  Calls.CONTENT_URI;

ContentResolver cr = getContentResolver();

Cursor cur = cr.query(llamadasUri,
        projection, //Columnas a devolver
        null,       //Condición de la query
        null,       //Argumentos variables de la query
        null);      //Orden de los resultados

if (cur.moveToFirst())
{
    int tipo;
    String tipoLlamada = "";
    String telefono;

    int colTipo = cur.getColumnIndex(Calls.TYPE);
    int colTelefono = cur.getColumnIndex(Calls.NUMBER);

    txtResultados.setText("");
```

```
    do
    {
        tipo = cur.getInt(colTipo);
        telefono = cur.getString(colTelefono);

        if(tipo == Calls.INCOMING_TYPE)
            tipoLlamada = "ENTRADA";
        else if(tipo == Calls.OUTGOING_TYPE)
            tipoLlamada = "SALIDA";
        else if(tipo == Calls.MISSED_TYPE)
            tipoLlamada = "PERDIDA";

        txtResultados.append(tipoLlamada + " - " + telefono + "\n");

    } while (cur.moveToNext());
}
```

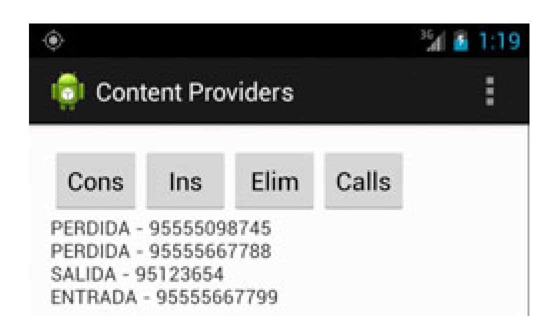
Lo único fuera de lo normal que hacemos en el código anterior es la decodificación del valor del tipo de llamada recuperado, que la hacemos comparando el resultado con las constantes `Calls.INCOMING_TYPE` (entrante), `Calls.OUTGOING_TYPE` (saliente), `Calls.MISSED_TYPE` (perdida) proporcionadas por la propia clase provider.

Un último detalle importante. Para que nuestra aplicación pueda acceder al historial de llamadas del dispositivo tendremos que incluir en el fichero `AndroidManifest.xml` los permisos `READ_CONTACTS` y `READ_CALL_LOG` utilizando la cláusula `<uses-permission>` correspondiente.

```
<uses-permission android:name="android.permission.READ_CONTACTS"></uses-permission>

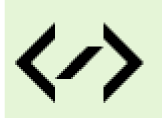
<uses-permission android:name="android.permission.READ_CALL_LOG"></uses-permission>
```

Si ejecutamos la aplicación y realizamos la consulta podremos ver un resultado similar al siguiente:



Y con esto terminamos con el tema dedicado a los *content providers*. Espero que os haya sido útil para aprender a incluir esta funcionalidad a vuestras aplicaciones y a utilizar este mecanismo para acceder a datos propios del sistema.

Podéis descargar el código fuente completo de la aplicación de ejemplo construida a lo largo de los dos apartados anteriores pulsando este enlace.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-content-providers-2

12

Notificaciones

XII. Notificaciones Android

Notificaciones Toast

En Android existen varias formas de notificar mensajes al usuario, como por ejemplo los cuadros de diálogo modales o las notificaciones de la bandeja del sistema (o barra de estado). Pero en este apartado nos vamos a centrar en primer lugar en la forma más sencilla de notificación: los llamados *Toast*.

Un *toast* es un mensaje que se muestra en pantalla durante unos segundos al usuario para luego volver a desaparecer automáticamente sin requerir ningún tipo de actuación por su parte, y sin recibir el foco en ningún momento (o dicho de otra forma, sin interferir en las acciones que esté realizando el usuario en ese momento). Aunque son personalizables, aparecen por defecto en la parte inferior de la pantalla, sobre un rectángulo gris ligeramente translúcido. Por sus propias características, este tipo de notificaciones son ideales para mostrar mensajes rápidos y sencillos al usuario, pero por el contrario, al no requerir confirmación por su parte, no deberían utilizarse para hacer notificaciones demasiado importantes.

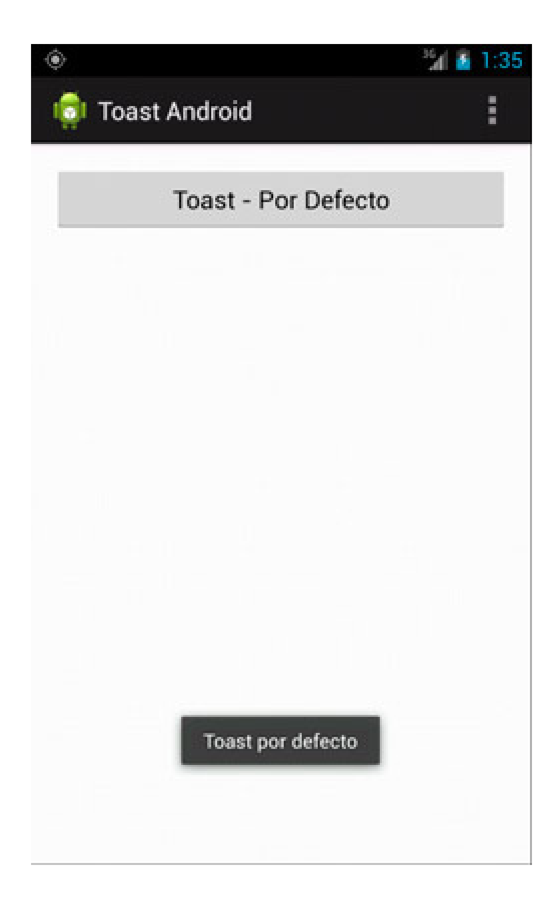
Su utilización es muy sencilla, concentrándose toda la funcionalidad en la clase `Toast`. Esta clase dispone de un método estático `makeText()` al que deberemos pasar como parámetro el contexto de la actividad, el texto a mostrar, y la duración del mensaje, que puede tomar los valores `LENGTH_LONG` o `LENGTH_SHORT`, dependiendo del tiempo que queramos que la notificación aparezca en pantalla. Tras obtener una referencia al objeto `Toast` a través de este método, ya sólo nos quedaría mostrarlo en pantalla mediante el método `show()`.

Vamos a construir una aplicación de ejemplo para demostrar el funcionamiento de este tipo de notificaciones. Y para empezar vamos a incluir un botón que muestre un toast básico de la forma que acabamos de describir:

```
btnDefecto.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        Toast toast1 =
            Toast.makeText(getApplicationContext(),
                    "Toast por defecto", Toast.LENGTH_SHORT);

        toast1.show();
    }
});
```

Si ejecutamos esta sencilla aplicación en el emulador y pulsamos el botón que acabamos de añadir veremos como en la parte inferior de la pantalla aparece el mensaje "`Toast por defecto`", que tras varios segundos desaparecerá automáticamente.



Como hemos comentado, éste es el comportamiento por defecto de las notificaciones toast, sin embargo también podemos personalizarlo un poco cambiando su posición en la pantalla. Para esto utilizaremos el método `setGravity()`, al que podremos indicar en qué zona deseamos que aparezca la notificación. La zona deberemos indicarla con alguna de las constantes definidas en la clase `Gravity`: `CENTER`, `LEFT`, `BOTTOM`, ... o con alguna combinación de éstas.

Para nuestro ejemplo vamos a colocar la notificación en la zona central izquierda de la pantalla. Para ello, añadamos un segundo botón a la aplicación de ejemplo que muestre un toast con estas características:

```
btnGravity.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        Toast toast2 =
            Toast.makeText(getApplicationContext(),
                    "Toast con gravity", Toast.LENGTH_SHORT);

        toast2.setGravity(Gravity.CENTER|Gravity.LEFT,0,0);

        toast2.show();
    }
});
```

Si volvemos a ejecutar la aplicación y pulsamos el nuevo botón veremos como el toast aparece en la zona indicada de la pantalla:



Si esto no es suficiente y necesitamos personalizar por completo el aspecto de la notificación, Android nos ofrece la posibilidad de definir un layout XML propio para toast, donde podremos incluir todos los elementos necesarios para adaptar la notificación a nuestras necesidades. Para nuestro ejemplo vamos a definir un layout sencillo, con una imagen y una etiqueta de texto sobre un rectángulo gris:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/lytLayout"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="horizontal"
  android:background="#555555"
  android:padding="5dip" >

    <ImageView android:id="@+id/imgIcono"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:src="@drawable/marcador" />

    <TextView android:id="@+id/txtMensaje"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:textColor="#FFFFFF"
        android:paddingLeft="10dip" />

</LinearLayout>
```

Guardaremos este layout con el nombre "`toast_layout.xml`", y como siempre lo colocaremos en la carpeta "`res\layout`" de nuestro proyecto.

Para asignar este layout a nuestro toast tendremos que actuar de una forma algo diferente a las anteriores. En primer lugar deberemos inflar el layout mediante un objeto LayoutInflater, como ya vimos en varias ocasiones al tratar los apartados de interfaz gráfica. Una vez construido el layout modificaremos los valores de los distintos controles para mostrar la información que queramos. En nuestro caso, tan sólo modificaremos el mensaje de la etiqueta de texto, ya que la imagen ya la asignamos de forma estática en el layout XML mediante el atributo android:src. Tras esto, sólo nos quedará establecer la duración de la notificación con setDuration() y asignar el layout personalizado al toast mediante el método setView(). Veamos cómo quedaría todo el código incluido en un tercer botón de ejemplo:

```
btnLayout.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        Toast toast3 = new Toast(getApplicationContext());

        LayoutInflater inflater = getLayoutInflater();
        View layout = inflater.inflate(R.layout.toast_layout,
                        (ViewGroup) findViewById(R.id.lytLayout));

        TextView txtMsg = (TextView)layout.findViewById(R.id.txtMensaje);
        txtMsg.setText("Toast Personalizado");

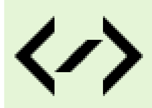
        toast3.setDuration(Toast.LENGTH_SHORT);
        toast3.setView(layout);
        toast3.show();
    }
});
```

Si ejecutamos ahora la aplicación de ejemplo y pulsamos el nuevo botón, veremos como nuestro toast aparece con la estructura definida en nuestro layout personalizado.



Como podéis comprobar, mostrar notificaciones de tipo Toast en nuestras aplicaciones Android es algo de lo más sencillo, y a veces resultan un elemento de lo más interesante para avisar al usuario de determinados

eventos.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

[curso-android-src/android-toast]

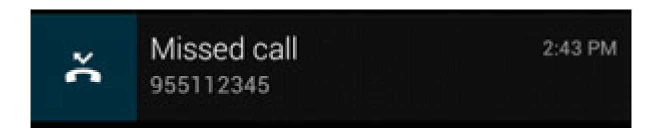
Notificaciones de la Barra de Estado

Hemos tratado ya un primer mecanismo de notificaciones disponibles en la plataforma Android: los llamados *Toast*. Como ya comentamos, este tipo de notificaciones, aunque resultan útiles y prácticas en muchas ocasiones, no deberíamos utilizarlas en situaciones en las que necesitemos asegurarnos la atención del usuario ya que no requiere de ninguna intervención por su parte, se muestran y desaparecen automáticamente de la pantalla.

En este nuevo apartado vamos a tratar otro tipo de notificaciones algo más persistentes, las notificaciones de la barra de estado de Android. Estas notificaciones son las que se muestran en nuestro dispositivo por ejemplo cuando recibimos un mensaje SMS, cuando tenemos actualizaciones disponibles, cuando tenemos el reproductor de música abierto en segundo plano, ... Estas notificaciones constan de un icono y un texto mostrado en la barra de estado superior, y adicionalmente un mensaje algo más descriptivo y una marca de fecha/hora que podemos consultar desplegando la bandeja del sistema. A modo de ejemplo, cuando tenemos una llamada perdida en nuestro terminal, se nos muestra por un lado un icono en la barra de estado superior (más un texto que aparece durante unos segundos)...

Missed call from 955112345

... y un mensaje con más información al desplegar la bandeja del sistema, donde en este caso concreto se nos informa del evento que se ha producido ("*Missed call*"), el número de teléfono asociado, y la fecha/hora del evento. Además, al pulsar sobre la notificación se nos dirige automáticamente al historial de llamadas.



Pues bien, aprendamos a utilizar este tipo de notificaciones en nuestras aplicaciones. Vamos a construir para ello una aplicación de ejemplo, como siempre lo más sencilla posible para centrar la atención en lo realmente importante. En este caso, el ejemplo va a consistir en un único botón que genere una notificación de ejemplo en la barra de estado, con todos los elementos comentados y con la posibilidad de dirigirnos a la propia aplicación de ejemplo cuando se pulse sobre ella.

Para generar notificaciones en la barra de estado del sistema vamos a utilizar una clase incluida en la librería de compatibilidad *android-support-v4.jar* que ya hemos utilizado en otras ocasiones y que debe estar incluida por defecto en vuestro proyecto si lo habéis creado con alguna versión reciente del plugin de Eclipse. Esto es así para asegurar la compatibilidad con versiones de Android antiguas, ya que las notificaciones son un elemento que han sufrido bastantes cambios en las versiones más recientes. La clase en cuestión se llama `NotificationCompat.Builder` y lo que tendremos que hacer será crear un nuevo objeto de este tipo pasándole el contexto de la aplicación y asignar todas las propiedades que queramos mediante sus métodos *set()*.

En primer lugar estableceremos los iconos a mostrar mediante los métodos `setSmallIcon()` y `setLargeIcon()` que se corresponden con los iconos mostrados a la derecha y a la izquierda del contenido de la notificación en versiones recientes de Android. En versiones más antiguas tan sólo se mostrará el icono pequeño a la izquierda de la notificación. Además, el icono pequeño también se mostrará en la barra de estado superior.

A continuación estableceremos el título y el texto de la notificación, utilizando para ello los métodos setContentTitle() y setContentText().

Por último, estableceremos el *ticker* (texto que aparece por unos segundos en la barra de estado al generarse una nueva notificación) mediante setTicker() y el texto auxiliar (opcional) que aparecerá a la izquierda del icono pequeño de la notificación mediante setContentInfo(). La fecha/hora asociada a nuestra notificación se tomará automáticamente de la fecha/hora actual si no se establece nada, o bien puede utilizarse el método setWhen() para indicar otra marca de tiempo. Veamos cómo quedaría nuestro código por el momento:

```
NotificationCompat.Builder mBuilder =
      new NotificationCompat.Builder(MainActivity.this)
            .setSmallIcon(android.R.drawable.stat_sys_warning)
            .setLargeIcon((((BitmapDrawable)getResources()
                  .getDrawable(R.drawable.ic_launcher)).getBitmap()))
            .setContentTitle("Mensaje de Alerta")
            .setContentText("Ejemplo de notificación.")
            .setContentInfo("4")
            .setTicker("Alerta!");
```

El segundo paso será establecer la actividad a la cual debemos dirigir al usuario automáticamente si éste pulsa sobre la notificación. Para ello debemos construir un objeto PendingIntent, que será el que contenga la información de la actividad asociada a la notificación y que será lanzado al pulsar sobre ella. Para ello definiremos en primer lugar un objeto Intent, indicando la clase de la actividad concreta a lanzar, que en nuestro caso será la propia actividad principal de ejemplo (MainActivity.class). Este *intent* lo utilizaremos para construir el PendingIntent final mediante el método PendingIntent.getActivity(). Por último asociaremos este objeto a la notificación mediante el método setContentIntent() del Builder. Veamos cómo quedaría esta última parte comentada:

```
Intent notIntent =
      new Intent(MainActivity.this, MainActivity.class);

PendingIntent contIntent = PendingIntent.getActivity(
                              MainActivity.this, 0, notIntent, 0);

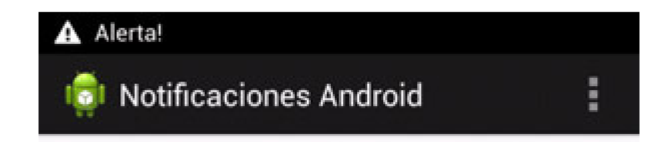
mBuilder.setContentIntent(contIntent);
```

Por último, una vez tenemos completamente configuradas las opciones de nuestra notificación podemos generarla llamando al método notify() del *Notification Manager*,al cual podemos acceder mediante una llamada a getSystemService() con la constante Context.NOTIFICATION_SERVICE. Por su parte al método notify() le pasaremos como parámetro un identificador único definido por nosotros que identifique a nuestra notificación y el resultado del builder que hemos construido antes, que obtenemos llamando a su método build().

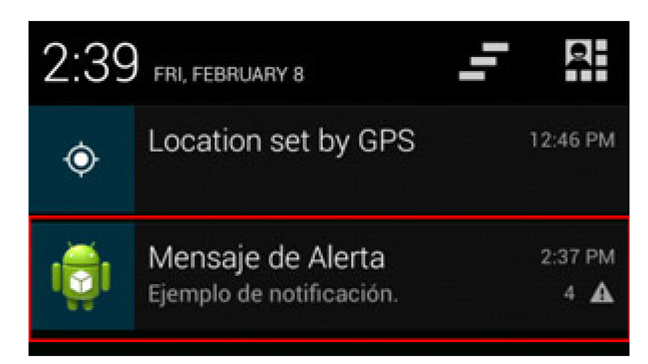
```
NotificationManager mNotificationManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);

mNotificationManager.notify(NOTIF_ALERTA_ID, mBuilder.build());
```

Ya estamos en disposición de probar nuestra aplicación de ejemplo. Para ello vamos a ejecutarla en el emulador de Android y pulsamos el botón que hemos implementado con todo el código anterior. Si todo va bien, debería aparecer en ese mismo momento nuestra notificación en la barra de estado, con el icono y texto definidos.

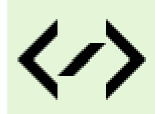


Si ahora salimos de la aplicación y desplegamos la bandeja del sistema podremos verificar el resto de información de la notificación tal como muestra la siguiente imagen:



Por último, si pulsamos sobre la notificación se debería abrir de nuevo automáticamente la aplicación de ejemplo.

En definitiva, como podéis comprobar es bastante sencillo generar notificaciones en la barra de estado de Android desde nuestras aplicaciones. Os animo a utilizar este mecanismo para notificar determinados eventos al usuario de forma bastante visual e intuitiva.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-notificaciones

Cuadros de Diálogo

Ya hemos visto dos de las principales alternativas a la hora de mostrar notificaciones a los usuarios de nuestras aplicaciones: los *toast* y las *notificaciones de la barra de estado*. En este último apartado sobre notificaciones vamos a comentar otro mecanismo que podemos utilizar para mostrar o solicitar información puntual al usuario. El mecanismo del que hablamos son los *cuadros de diálogo*.

En principio, los *diálogos* de Android los podremos utilizar con distintos fines, en general:

- Mostrar un mensaje.
- Pedir una confirmación rápida.
- Solicitar al usuario una elección (simple o múltiple) entre varias alternativas.

De cualquier forma, veremos también cómo personalizar completamente un diálogo para adaptarlo a cualquier otra necesidad.

El uso actual de los diálogos en Android se basa en fragmets, pero por suerte tenemos toda la funcionalidad implementada una vez más en la librería de compatibilidad *android-support-v4.jar* (que debe estar incluida por defecto en tu proyecto si lo has creado con una versión reciente del pluin de Eclipse) por lo que no tendremos problemas al ejecutar nuestra aplicación en versiones antiguas de Android. En este caso nos vamos a basar en la clase `DialogFragment`. Para crear un diálogo lo primero que haremos será crear una nueva clase que herede de DialogFragment y sobrescribiremos uno de sus métodos, `onCreateDialog()`, que será el encargado de construir el diálogo con las opciones que necesitemos.

La forma de construir cada diálogo dependerá de la información y funcionalidad que necesitemos. A continuación mostraré algunas de las formas más habituales.

Diálogo de Alerta

Este tipo de diálogo se limita a mostrar un mensaje sencillo al usuario, y un único botón de OK para confirmar su lectura. Lo construiremos mediante la clase `AlertDialog`, y más concretamente su subclase `AlertDialog.Builder`, de forma similar a las notificaciones de barra de estado que ya hemos comentado en el capítulo anterior. Su utilización es muy sencilla, bastará con crear un objeto de tipo `AlertDialog.Builder` y establecer las propiedades del diálogo mediante sus métodos correspondientes: título [`setTitle()`], mensaje [`setMessage()`] y el texto y comportamiento del botón [`setPositiveButton()`]. Veamos un ejemplo:

```
public class DialogoAlerta extends DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {

        AlertDialog.Builder builder =
                  new AlertDialog.Builder(getActivity());

        builder.setMessage("Esto es un mensaje de alerta.")
             .setTitle("Información")
             .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                   public void onClick(DialogInterface dialog, int id) {
                     dialog.cancel();
                   }
               });

        return builder.create();
    }
}
```

Como vemos, al método `setPositiveButton()` le pasamos como argumentos el texto a mostrar en el botón, y la implementación del evento onClick en forma de objeto `OnClickListener`. Dentro de este evento, nos limitamos a cerrar el diálogo mediante su método `cancel()`, aunque podríamos realizar cualquier otra acción.

Para lanzar este diálogo por ejemplo desde nuestra actividad principal, obtendríamos una referencia al Fragment Manager mediante una llamada a getSupportFragmentManager(), creamos un nuevo objeto de tipo DialogoAlerta y por último mostramos el diálogo mediante el método show() pasándole la referencia al fragment manager y una etiqueta identificativa del diálogo.

```
btnAlerta.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
            FragmentManager fragmentManager = getSupportFragmentManager();
             DialogoAlerta dialogo = new DialogoAlerta();
            dialogo.show(fragmentManager, "tagAlerta");
      }
});
```

El aspecto de nuestro diálogo de alerta sería el siguiente:

Información

Esto es un mensaje de alerta.

OK

Diálogo de Confirmación

Un diálogo de confirmación es muy similar al anterior, con la diferencia de que lo utilizaremos para solicitar al usuario que nos confirme una determinada acción, por lo que las posibles respuestas serán del tipo Sí/No.

La implementación de estos diálogos será prácticamente igual a la ya comentada para las alertas, salvo que en esta ocasión añadiremos dos botones, uno de ellos para la respuesta afirmativa (`setPositiveButton()`), y el segundo para la respuesta negativa (`setNegativeButton()`). Veamos un ejemplo:

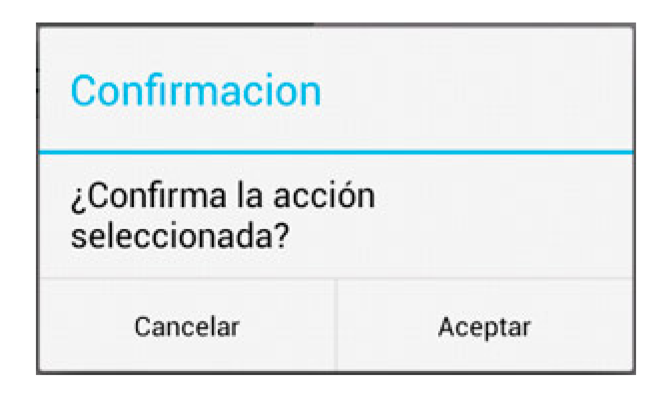
```
public class DialogoConfirmacion extends DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {

        AlertDialog.Builder builder =
                  new AlertDialog.Builder(getActivity());

        builder.setMessage("¿Confirma la acción seleccionada?")
        .setTitle("Confirmacion")
        .setPositiveButton("Aceptar", new DialogInterface.OnClickListener()  {
               public void onClick(DialogInterface dialog, int id) {
                        Log.i("Dialogos", "Confirmacion Aceptada.");
                                    dialog.cancel();
                   }
               })
        .setNegativeButton("Cancelar", new DialogInterface.OnClickListener() {
               public void onClick(DialogInterface dialog, int id) {
                        Log.i("Dialogos", "Confirmacion Cancelada.");
                                    dialog.cancel();
                   }
               });

        return builder.create();
    }
}
```

En este caso, generamos a modo de ejemplo dos mensajes de log para poder verificar qué botón pulsamos en el diálogo. El aspecto visual de nuestro diálogo de confirmación sería el siguiente:



Diálogo de Selección

Cuando las opciones a seleccionar por el usuario no son sólo dos, como en los diálogos de confirmación, sino que el conjunto es mayor podemos utilizar los diálogos de selección para mostrar una lista de opciones entre las que el usuario pueda elegir.

Para ello también utilizaremos la clase `AlertDialog`, pero esta vez no asignaremos ningún mensaje ni definiremos las acciones a realizar por cada botón individual, sino que directamente indicaremos la lista de opciones a mostrar (mediante el método `setItems()`) y proporcionaremos la implementación del evento `onClick()` sobre dicha lista (mediante un listener de tipo `DialogInterface.OnClickListener`), evento en el que realizaremos las acciones oportunas según la opción elegida. La lista de opciones la definiremos como un array tradicional. Veamos cómo:

```
public class DialogoSeleccion extends DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {

       final String[] items = {"Español", "Inglés", "Francés"};

        AlertDialog.Builder builder =
                  new AlertDialog.Builder(getActivity());

        builder.setTitle("Selección")
           .setItems(items, new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int item) {
                    Log.i("Dialogos", "Opción elegida: " + items[item]);
                }
            });

        return builder.create();
    }
}
```

En este caso el diálogo tendrá un aspecto similar a la interfaz mostrada para los controles `Spinner`.



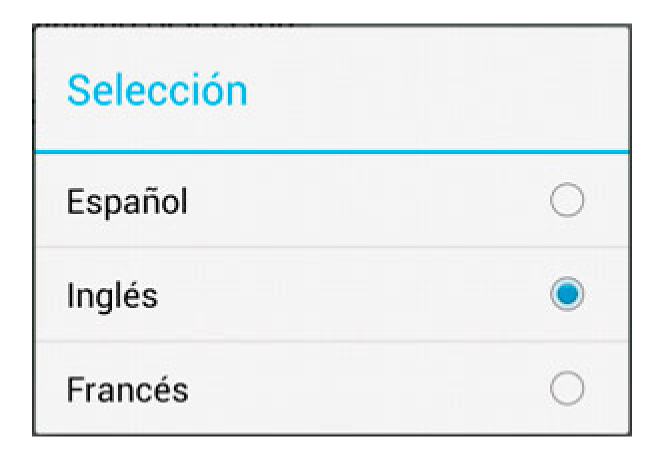
Este diálogo permite al usuario elegir entre las opciones disponibles cada vez que se muestra en pantalla.

Pero, ¿y si quisiéramos recordar cuál es la opción u opciones seleccionadas por el usuario para que aparezcan marcadas al visualizar de nuevo el cuadro de diálogo? Para ello podemos utilizar los métodos `setSingleChoiceItems()` o `setMultiChiceItems()`, en vez de el `setItems()` utilizado anteriormente. La diferencia entre ambos métodos, tal como se puede suponer por su nombre, es que el primero de ellos permitirá una selección simple y el segundo una selección múltiple, es decir, de varias opciones al mismo tiempo, mediante controles `CheckBox`.

La forma de utilizarlos es muy similar a la ya comentada. En el caso de `setSingleChoiceItems()`, el método tan sólo se diferencia de `setItems()` en que recibe un segundo parámetro adicional que indica el índice de la opción marcada por defecto. Si no queremos tener ninguna de ellas marcadas inicialmente pasaremos el valor `-1`.

```
builder.setTitle("Selección")
       .setSingleChoiceItems(items, -1,
               new DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int item) {
               Log.i("Dialogos", "Opción elegida: " + items[item]);
           }
       });
```

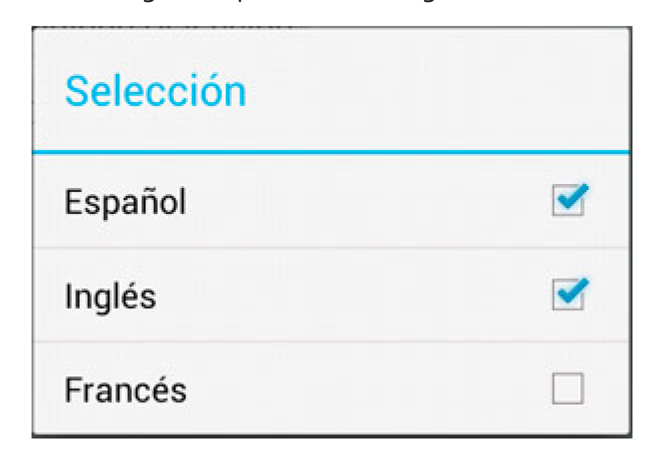
De esta forma conseguiríamos un diálogo como el de la siguiente imagen:



Si por el contrario optamos por la opción de selección múltiple, la diferencia principal estará en que tendremos que implementar un listener del tipo `DialogInterface.OnMultiChoiceClickListener`. En este caso, en el evento `onClick` recibiremos tanto la opción seleccionada (item) como el estado en el que ha quedado (`isChecked`). Además, en esta ocasión, el segundo parámetro adicional que indica el estado por defecto de las opciones ya no será un simple número entero, sino que tendrá que ser un array de booleanos. En caso de no querer ninguna opción seleccionada por defecto pasaremos el valor `null`.

```
builder.setTitle("Selección")
    .setMultiChoiceItems(items, null,
      new DialogInterface.OnMultiChoiceClickListener() {
            public void onClick(DialogInterface dialog, int item, boolean
isChecked) {
             Log.i("Dialogos", "Opción elegida: " + items[item]);
       }
});
```

Y el diálogo nos quedaría de la siguiente forma:



Tanto si utilizamos la opción de selección simple como la de selección múltiple, para salir del diálogo tendremos que pulsar la tecla "*Atrás*" de nuestro dispositivo.

Diálogos Personalizados

Por último, vamos a comentar cómo podemos establecer completamente el aspecto de un cuadro de diálogo. Para esto vamos a actuar como si estuviéramos definiendo la interfaz de una actividad, es decir, definiremos un layout XML con los elementos a mostrar en el diálogo. En mi caso voy a definir un layout de ejemplo llamado `dialog_personal.xml` que colocaré como siempre en la carpeta `res/layout`. Contendrá por ejemplo una imagen a la izquierda y dos líneas de texto a la derecha:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    android:padding="3dp" >

    <ImageView
        android:id="@+id/imageView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_launcher" />
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="3dp">
    <TextView
            android:id="@+id/textView1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/dialogo_linea_1" />
```

```
        <TextView
            android:id="@+id/textView2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/dialogo_linea_2" />

    </LinearLayout>

</LinearLayout>
```

Por su parte, en el método `onCreateDialog()` correspondiente utilizaremos el método `setView()` del builder para asociarle nuestro layout personalizado, que previamente tendremos que inflar como ya hemos comentado otras veces utilizando el método `inflate()`. Finalmente podremos incluir botones tal como vimos para los diálogos de alerta o confirmación. En este caso de ejemplo incluiremos un botón de *Aceptar*.

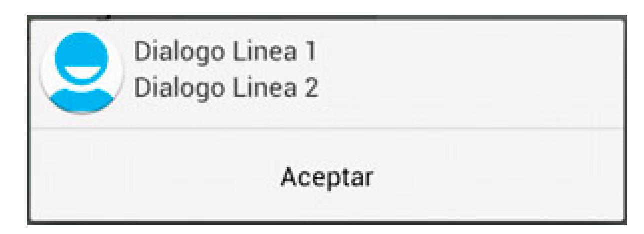
```
public class DialogoPersonalizado extends DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {

      AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
      LayoutInflater inflater = getActivity().getLayoutInflater();

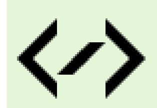
      builder.setView(inflater.inflate(R.layout.dialog_personal, null))
         .setPositiveButton("Aceptar", new DialogInterface.OnClickListener() {
             public void onClick(DialogInterface dialog, int id) {
                    dialog.cancel();
             }
      });

      return builder.create();
    }
}
```

De esta forma, si ejecutamos de nuevo nuestra aplicación de ejemplo y lanzamos el diálogo personalizado veremos algo como lo siguiente:



Y con esto terminamos con el apartado dedicado a notificaciones en Android. Hemos comentado los tres principales mecanismos de Android a la hora de mostrar mensajes y notificaciones al usuario (Toast, Barra de Estado, y Diálogos), todos ellos muy útiles para cualquier tipo de aplicación y que es importante conocer bien para poder decidir entre ellos dependiendo de las necesidades que tengamos.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-dialogos

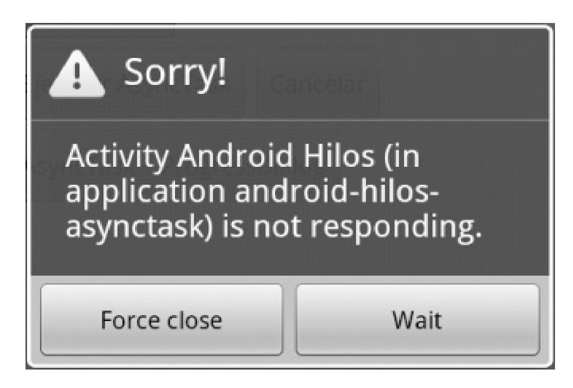
13

Tareas en Segundo Plano

XIII. Tareas en Segundo Plano

Hilos y Tareas Asíncronas (Thread y AsyncTask)

Todos los componentes de una aplicación Android, tanto las actividades, los servicios [sí, también los servicios], o los broadcast receivers se ejecutan en el mismo hilo de ejecución, el llamado *hilo principal*, *main thread* o *GUI thread*, que como éste último nombre indica también es el hilo donde se ejecutan todas las operaciones que gestionan la interfaz de usuario de la aplicación. Es por ello, que cualquier operación larga o costosa que realicemos en este hilo va a bloquear la ejecución del resto de componentes de la aplicación y por supuesto también la interfaz, produciendo al usuario un efecto evidente de lentitud, bloqueo, o mal funcionamiento en general, algo que deberíamos evitar a toda costa. Incluso puede ser peor, dado que Android monitoriza las operaciones realizadas en el hilo principal y detecta aquellas que superen los 5 segundos, en cuyo caso se muestra el famoso mensaje de "*Application Not Responding*" (ANR) y el usuario debe decidir entre forzar el cierre de la aplicación o esperar a que termine.



Obviamente, éstos son el tipo de errores que nadie quiere ver al utilizar su aplicación, y en este apartado y los siguientes vamos a ver varias formas de evitarlo utilizando procesos en segundo plano para ejecutar las operaciones de larga duración. En este primer apartado de la serie nos vamos a centrar en dos de las alternativas más directas a la hora de ejecutar tareas en segundo plano en Android:

- Crear nosotros mismos de forma explícita un nuevo hilo para ejecutar nuestra tarea.
- Utilizar la clase auxiliar `AsyncTask` proporcionada por Android.

Mi idea inicial para este capítulo era obviar la primera opción, ya que normalmente la segunda solución nos es más que suficiente, y además es mas sencilla y más limpia de implementar. Sin embargo, si no comentamos al menos de pasada la forma de crear "a mano" nuevos hilos y los problemas que surgen, quizá no se viera demasiado claro las ventajas que tiene utilizar las AsyncTask. Por tanto, finalmente voy a pasar muy rápidamente por la primera opción para después centrarnos un poco más en la segunda. Además, aprovechando el tema de la ejecución de tareas en segundo plano, vamos a ver también cómo utilizar un control (el ProgressBar) y un tipo de diálogo (el `ProgressDialog`) que no vimos en los primeros temas del curso dedicados a la interfaz de usuario.

Y para ir paso a paso, vamso a empezar por crear una aplicación de ejemplo en cuya actividad principal colocaremos un control `ProgressBar` (en mi caso llamado `pbarProgreso`) y un botón (`btnSinHilos`) que ejecute una tarea de larga duración. Para simular una operación de larga duración vamos a ayudarnos de un método auxiliar que lo único que haga sea esperar 1 segundo, mediante una llamada a `Thread.sleep()`.

```
private void tareaLarga()
{
    try {
        Thread.sleep(1000);
    } catch(InterruptedException e) {}
}
```

Haremos que nuestro botón ejecute este método 10 veces, de forma que nos quedará una ejecución de unos 10 segundos en total:

```
btnSinHilos.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        pbarProgreso.setMax(100);
        pbarProgreso.setProgress(0);

        for(int i=1; i<=10; i++) {
            tareaLarga();
            pbarProgreso.incrementProgressBy(10);
        }

        Toast.makeText(MainHilos.this, "Tarea finalizada!",
                Toast.LENGTH_SHORT).show();
    }
});
```

Como veis, aquí todavía no estamos utilizando nada especial, por lo que todo el código se ejecutará en el hilo principal de la aplicación. En cuanto a la utilización del control `ProgressBar` vemos que es muy sencilla y no requiere apenas configuración. En nuestro caso tan sólo hemos establecido el valor máximo que alcanzará (el valor en el que la barra de progreso estará rellena al máximo) mediante el método `setMax(100)`, posteriormente la hemos inicializado al valor mínimo mediante una llamada a `setProgress(0)` de forma que inicialmente aparezca completamente vacía, y por último en cada iteración del bucle incrementamos su valor en 10 unidades llamando a `incrementProgressBy(10)`, de tal forma que tras la décima iteración la barra llegue al valor máximo de 100 que establecimos antes. Finalmente mostramos un mensaje `Toast` para informar de la finalización de la tarea. Pues bien, ejecutemos la aplicación, pulsemos el botón, y veamos qué ocurre.

He colocado un pequeño vídeo al final del capítulo donde puede verse el resultado final de todas las pruebas que haremos durante este tutorial. En concreto esta primera prueba puede verse entre los segundos 00:00 – 00:12

No era eso lo que esperábamos ¿verdad? Lo que ha ocurrido es que desde el momento que hemos pulsado el botón para ejecutar la tarea, hemos bloqueado completamente el resto de la aplicación, incluida la actualización de la interfaz de usuario, que ha debido esperar a que ésta termine mostrando directamente la barra de progreso completamente llena. En definitiva, no hemos sido capaces de ver el progreso de la tarea. Pero como decíamos, este efecto puede empeorar. Probemos ahora a pulsar el botón de la tarea y mientras ésta se ejecuta realicemos cualquier acción sobre la pantalla, un simple click sobre el fondo nos basta. Veamos qué ocurre ahora.

Puedes verlo en el vídeo entre los segundos 00:13 – 00:28

Vemos cómo al intentar hacer cualquier acción sobre la aplicación Android nos ha advertido con un mensaje de error que la aplicación no responde debido a que se está ejecutando una operación de larga duración en el hilo principal. El usuario debe elegir entre esperar a que termine de ejecutarla o forzar el cierre de la aplicación. Pues bien, estos son los efectos que vamos a intentar evitar. La opción más inmediata que nos proporciona Android, al igual que otras plataformas, es crear directamente hilos secundarios dentro de los cuales ejecutar nuestras operaciones costosas. Esto lo conseguimos en Android instanciando un objeto de la clase `Thread`. El constructor de la clase Thread recibe como parámetro un nuevo objeto Runnable que

debemos construir implementando su método `run()`, dentro del cual vamos a realizar nuestra tarea de larga duración. Hecho esto, debemos llamar al método `start()` del objeto Threaddefinido para comenzar la ejecución de la tarea en segundo plano.

```
new Thread(new Runnable() {
    public void run() {
        //Aquí ejecutamos nuestras tareas costosas
    }
}).start();
```

Hasta aquí todo sencillo y relativamente limpio. Los problemas aparecen cuando nos damos cuenta que desde este hilo secundario que hemos creado no podemos hacer referencia directa a componentes que se ejecuten en el hilo principal, entre ellos los controles que forman nuestra interfaz de usuario, es decir, que desde el método `run()` no podríamos ir actualizando directamente nuestra barra de progreso de la misma forma que lo hacíamos antes. Para solucionar esto, Android proporciona varias alternativas, entre ellas la utilización del método `post()` para actuar sobre cada control de la interfaz, o la llamada al método `runOnUiThread()` para "enviar" operaciones al hilo principal desde el hilo secundario [Nota: Sí, vale, sé que no he nombrado la opción de los `Handler`, pero no quería complicar más el tema por el momento]. Ambas opciones requieren como parámetro un nuevo objeto Runnable del que nuevamente habrá que implementar su método `run()` donde se actúe sobre los elementos de la interfaz. Por ver algún ejemplo, en nuestro caso hemos utilizado el método `post()` para actuar sobre el control `ProgressBar`, y el método `runOnUiThread()` para mostrar el mensaje toast.

```
new Thread(new Runnable() {
    public void run() {
        pbarProgreso.post(new Runnable() {
        public void run() {
            pbarProgreso.setProgress(0);
        }
    });

    for(int i=1; i<=10; i++) {
        tareaLarga();
        pbarProgreso.post(new Runnable() {
            public void run() {
                pbarProgreso.incrementProgressBy(10);
                }
            });
    }

    runOnUiThread(new Runnable() {
        public void run() {
            Toast.makeText(MainHilos.this, "Tarea finalizada!",
                Toast.LENGTH_SHORT).show();
        }
    });
    }
}).start();
```

Utilicemos este código dentro de un nuevo botón de nuestra aplicación de ejemplo y vamos a probarlo en el emulador.

Puedes verlo en el vídeo entre los segundos 00:29 – 00:43

Ahora sí podemos ver el progreso de nuestra tarea reflejado en la barra de progreso. La creación de un hilo secundario nos ha permitido mantener el hilo principal libre de forma que nuestra interfaz de usuario de actualiza sin problemas durante la ejecución de la tarea en segundo plano. Sin embargo miremos de nuevo

el código anterior. Complicado de leer, ¿verdad? Y eso considerando que tan sólo estamos actualizando un control de nuestra interfaz. Si el número de controles fuera mayor, o necesitáramos una mayor interacción con la interfaz el código empezaría a ser inmanejable, difícil de leer y mantener, y por tanto también más propenso a errores. Pues bien, aquí es donde Android llega en nuestra ayuda y nos ofrece la clase `AsyncTask`, que nos va a permitir realizar esto mismo pero con la ventaja de no tener que utilizar artefactos del tipo `runOnUiThread()` y de una forma mucho más organizada y legible. La forma básica de utilizar la clase `AsyncTask` consiste en crear una nueva clase que extienda de ella y sobrescribir varios de sus métodos entre los que repartiremos la funcionalidad de nuestra tarea. Estos métodos son los siguientes:

- `onPreExecute()`. Se ejecutará antes del código principal de nuestra tarea. Se suele utilizar para preparar la ejecución de la tarea, inicializar la interfaz, etc.
- `doInBackground()`. Contendrá el código principal de nuestra tarea.
- `onProgressUpdate()`. Se ejecutará cada vez que llamemos al método `publishProgress()` desde el método `doInBackground()`.
- `onPostExecute()`. Se ejecutará cuando finalice nuestra tarea, o dicho de otra forma, tras la finalización del método `doInBackground()`.
- `onCancelled()`. Se ejecutará cuando se cancele la ejecución de la tarea antes de su finalización normal.

Estos métodos tienen una particularidad esencial para nuestros intereses. El método `doInBackground()` se ejecuta en un hilo secundario (por tanto no podremos interactuar con la interfaz), pero sin embargo todos los demás se ejecutan en el hilo principal, lo que quiere decir que dentro de ellos podremos hacer referencia directa a nuestros controles de usuario para actualizar la interfaz. Por su parte, dentro de `doInBackground()` tendremos la posibilidad de llamar periódicamente al método `publishProgress()` para que automáticamente desde el método `onProgressUpdate()` se actualice la interfaz si es necesario. Al extender una nueva clase de `AsyncTask` indicaremos tres parámetros de tipo:

- El tipo de datos que recibiremos como **entrada** de la tarea en el método `doInBackground()`.
- El tipo de datos con el que actualizaremos el **progreso** de la tarea, y que recibiremos como parámetro del método `onProgressUpdate()` y que a su vez tendremos que incluir como parámetro del método `publishProgress()`.
- El tipo de datos que devolveremos como **resultado** de nuestra tarea, que será el tipo de retorno del método `doInBackground()` y el tipo del parámetro recibido en el método `onPostExecute()`.

En nuestro caso de ejemplo, extenderemos de `AsyncTask` indicando los tipos `Void`, `Integer` y `Boolean` respectivamente, lo que se traducirá en que:

- `doInBackground()` no recibirá ningún parámetro de entrada (Void).
- `publishProgress()` y `onProgressUpdate()` recibirán como parámetros datos de tipo entero (`Integer`).
- `doInBackground()` devolverá como retorno un dato de tipo booleano y `onPostExecute()` también recibirá como parámetro un dato del dicho tipo (`Boolean`).

Dicho esto, cómo repartiremos la funcionalidad de nuestra tarea entre los distintos métodos. Pues sencillo, en `onPreExecute()` inicializaremos la barra de progreso estableciendo su valor máximo y poniéndola a cero para comenzar. En `doInBackground()` ejecutaremos nuestro bucle habitual llamando a publishProgress() tras cada iteración indicando el progreso actual. En `onProgressUpdate()` actualizaremos el estado de la barra de progreso con el valor recibido como parámetro. Y por último en `onPostExecute()` mostraremos el mensaje `Toast` de finalización de la tarea. Veamos el código completo:

```
private class MiTareaAsincrona extends AsyncTask<Void, Integer, Boolean> {

    @Override
    protected Boolean doInBackground(Void... params) {

        for(int i=1; i<=10; i++) {
            tareaLarga();

            publishProgress(i*10);

            if(isCancelled())
                break;
        }

        return true;
    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        int progreso = values[0].intValue();

        pbarProgreso.setProgress(progreso);
    }

    @Override
    protected void onPreExecute() {
        pbarProgreso.setMax(100);
        pbarProgreso.setProgress(0);
    }

    @Override
    protected void onPostExecute(Boolean result) {
        if(result)
            Toast.makeText(MainHilos.this, "Tarea finalizada!",
                    Toast.LENGTH_SHORT).show();
    }

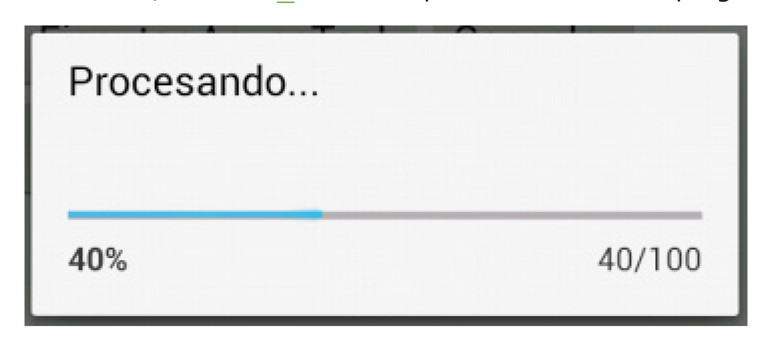
    @Override
    protected void onCancelled() {
        Toast.makeText(MainHilos.this, "Tarea cancelada!",
                Toast.LENGTH_SHORT).show();
    }
}
```

Si observamos con detenimiento el código, la única novedad que hemos introducido es la posibilidad de cancelar la tarea en medio de su ejecución. Esto se realiza llamando al método `cancel()` de nuestra `AsyncTask` (para lo cual añadiremos un botón más a nuestra aplicación de ejemplo, además del nuevo que añadiremos para comenzar la tarea). Dentro de la ejecución de nuestra tarea en `doInBackground()` tendremos además que consultar periodicamente el resultado del método `isCancelled()` que nos dirá si el usuario ha cancelado la tarea (es decir, si se ha llamado al método `cancel()`), en cuyo caso deberemos de terminar la ejecución lo antes posible, en nuestro caso de ejemplo simplemente saldremos del bucle con la instrucción `break`. Además, tendremos en cuenta que en los casos que se cancela la tarea, tras el método `doInBackground()` no se llamará a `onPostExecute()` sino al método `onCancelled()`, dentro del cual podremos realizar cualquier acción para confirma la cancelación de la tarea. En nuestro caso mostraremos un mensaje Toast informando de ello.

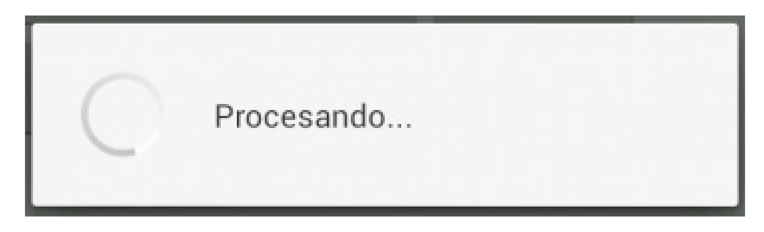
Puedes verlo en el vídeo entre los segundos 00:44 – 01:06

Mucho mejor que las alternativas anteriores, verdad? Pero vamos a mostrar una opción más. Si queremos que el usuario pueda ver el progreso de nuestra tarea en segundo plano, pero no queremos que interactúe mientras tanto con la aplicación tenemos la opción de mostrar la barra de progreso dentro de un diálogo. Android nos proporciona directamente un componente de este tipo en forma de un tipo especial de diálogo llamado `ProgressDialog`.

Configurar un cuadro de diálogo de este tipo es muy sencillo. Para ello vamos a añadir un botón más a nuestra aplicación de ejemplo, donde inicializaremos el diálogo y lanzaremos la tarea en segundo plano. Para inicializar el diálogo comenzaremos por crear un nuevo objeto ProgressDialog pasándole como parámetro el contexto actual. Tras esto estableceremos su estilo: `STYLE_HORIZONTAL` para una barra de progreso tradicional, o `STYLE_SPINNER` para un indicador de progreso de tipo indeterminado.



ProgressDialog horizontal



ProgressDialog spinner

Lo siguiente será especificar el texto a mostrar en el diálogo, en nuestro caso el mensaje "Procesando...", y el valor máximo de nuestro progreso, que lo mantendremos en 100. Por último indicaremos si deseamos que el diálogo sea cancelable, es decir, que el usuario pueda cerrarlo pulsando el botón Atrás del teléfono. Para nuestro ejemplo activaremos esta propiedad para ver cómo podemos cancelar también nuestra tarea en segundo plano cuando el usuario cierra el diálogo. Tras la configuración del diálogo lanzaremos la `AsyncTask` del ejemplo anterior, que tendremos que modificar ligeramente para adaptarla al nuevo diálogo. Veamos el código por ahora:

```
btnAsyncDialog.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View v) {

        pDialog = new ProgressDialog(MainHilos.this);
        pDialog.setProgressStyle(ProgressDialog.STYLE_SPINNER);
        pDialog.setMessage("Procesando...");
        pDialog.setCancelable(true);
        pDialog.setMax(100);

        tarea2 = new MiTareaAsincronaDialog();
        tarea2.execute();
    }
});
```

La `AsyncTask` será muy similar a la que ya implementamos. De hecho el método `doInBackground()`

no sufrirá cambios.

En onProgressUpdate() la única diferencia será que actualizaremos el progreso llamando al método setProgress() del ProgressDialog en vez de la ProgressBar.

El código de onPreExecute() sí tendrá algún cambio más. Aprovecharemos este método para implementar el evento onCancel del diálogo, dentro del cual cancelaremos también la tarea en segundo plano llamando al método cancel(). Además, inicializaremos el progreso del diálogo a 0 y lo mostraremos al usuario mediante el método show().

Por último, en el método onPostExecute() además de mostrar el Toast de finalización, tendremos que cerrar previamente el diálogo llamando a su método dismiss().

Veamos el código completo de la AsyncTask modificada para usar el nuevo ProgressDialog.

```
private class MiTareaAsincronaDialog extends AsyncTask<Void, Integer, Boolean>
{
        @Override
        protected Boolean doInBackground(Void... params) {

            for(int i=1; i<=10; i++) {
            tareaLarga();

            publishProgress(i*10);

            if(isCancelled())
                break;
        }

            return true;
        }

        @Override
        protected void onProgressUpdate(Integer... values) {
            int progreso = values[0].intValue();

            pDialog.setProgress(progreso);
        }

        @Override
        protected void onPreExecute() {

            pDialog.setOnCancelListener(new OnCancelListener() {
                @Override
                public void onCancel(DialogInterface dialog) {
                    MiTareaAsincronaDialog.this.cancel(true);
                }
            });

            pDialog.setProgress(0);
            pDialog.show();
        }
```

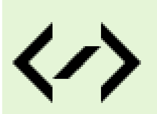
```
        @Override
        protected void onPostExecute(Boolean result) {
            if(result)
            {
                pDialog.dismiss();
                Toast.makeText(MainHilos.this, "Tarea finalizada!",
                    Toast.LENGTH_SHORT).show();
            }
        }

        @Override
        protected void onCancelled() {
            Toast.makeText(MainHilos.this, "Tarea cancelada!",
                Toast.LENGTH_SHORT).show();
        }
    }
```

Si ahora ejecutamos nuestro proyecto y pulsamos sobre el último botón incluido veremos cómo el diálogo aparece por encima de nuestra actividad mostrando el progreso de la tarea asíncrona. Tras finalizar, el diálogo desaparece y se muestra el mensaje toast de finalización. Si en cambio, se pulsa el botón Atrás del dispositivo antes de que la tarea termine el diálogo se cerrará y se mostrará el mensaje de cancelación.

Puedes verlo en el vídeo entre los segundos 01:07 – 01:35

Y con esto habríamos concluido este primer apartado sobre hilos y tareas en segundo plano. Os dejo a continuación el vídeo de demostración de la aplicación de ejemplo construida durante el tema: http://www.youtube.com/watch?v=tSia91rGaCo



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-asynctask

IntentService

En el apartado anterior vimos cómo ejecutar tareas en segundo plano haciendo uso de hilos (`Thread`) y tareas asíncronas (`AsyncTask`). En este nuevo apartado nos vamos a centrar en una alternativa menos conocida, aunque tanto o más interesante, para conseguir el mismo objetivo: ejecutar una determinada tarea en un hilo independiente al hilo principal de la aplicación. Esta opción se llama `IntentService`, y no es más que un tipo particular de servicio Android que se preocupará por nosotros de la creación y gestión del nuevo hilo de ejecución y de detenerse a sí mismo una vez concluida su tarea asociada.

Como en el caso de las `AsyncTask`, la utilización de un `IntentService` va a ser tan sencilla como extender una nueva clase de `IntentService` e implementar su método `onHandleIntent()`. Este método recibe como parámetro un Intent, que podremos utilizar para pasar al servicio los datos de entrada necesarios.

A diferencia de las `AsyncTask`, un `IntentService` no proporciona métodos que se ejecuten en el hilo principal de la aplicación y que podamos aprovechar para "comunicarnos" con nuestra interfaz durante la ejecución. Éste es el motivo principal de que los `IntentService` sean una opción menos utilizada a la hora de ejecutar tareas que requieran cierta vinculación con la interfaz de la aplicación. Sin embargo tampoco es imposible su uso en este tipo de escenarios ya que podremos utilizar por ejemplo mensajes broadcast (y por supuesto su `BroadcastReceiver` asociado capaz de procesar los mensajes) para comunicar eventos al hilo principal, como por ejemplo la necesidad de actualizar controles de la interfaz o simplemente para comunicar la finalización de la tarea ejecutada. En este apartado veremos cómo implementar este método

para conseguir una aplicación de ejemplo similar a la que construimos en el apartado anterior utilizando `AsyncTask`.

Empezaremos extendiendo una nueva clase derivada de IntentService, que para ser originales llamaremos `MiIntentService`. Lo primero que tendremos que hacer será implementar un constructor por defecto. Este constructor lo único que hará será llamar al constructor de la clase padre pasándole el nombre de nuestra nueva clase.

A continuación implementaremos el método `onHandleIntent()`. Como ya hemos indicado, este método será el que contenga el código de la tarea a ejecutar en segundo plano. Para simular una tarea de larga duración utilizaremos el mismo bucle que ya vimos en el apartado anterior con la novedad de que esta vez el número de iteraciones será variable, de forma que podamos experimentar con cómo pasar datos de entrada a través del Intent recibido como parámetro en `onHandleIntent()`. En nuestro caso de ejemplo pasaremos un sólo dato de entrada que indique el número de iteraciones. Por tanto, lo primero que haremos será obtener este dato a partir del Intent mediante el método `getIntExtra()`. Una vez conocemos el número de iteraciones, tan sólo nos queda ejecutar el bucle y comunicar el progreso tras cada iteración.

Como ya hemos comentado, para comunicar este progreso vamos a hacer uso de mensajes broadcast. Para enviar este tipo de mensajes necesitamos construir un Intent, asociarle un nombre de acción determinada que lo identifique mediante `setAction()`, e incluir los datos que necesitemos comunicar añadiendo tantos extras como sean necesarios mediante el método `putExtra()`. Los nombres de las acciones se suelen preceder con el paquete java de nuestra aplicación de forma que nos aseguremos que es un identificador único. En nuestro caso lo llamaremos "`net.sgoliver.intent.action.PROGRESO`" y lo definiremos como atributo estático de la clase para mayor comodidad, llamado `ACTION_PROGRESO`. Por su parte, los datos a comunicar en nuestro ejemplo será solo el nivel de progreso, por lo que sólo añadiremos un extra a nuestro intent con dicho dato. Por último enviaremos el mensaje llamando al método `sendBroadcast()` pasándole como parámetro el intent recién creado. Veamos cómo quedaría el código completo.

```
public class MiIntentService extends IntentService {

    public static final String ACTION_PROGRESO =
        "net.sgoliver.intent.action.PROGRESO";
    public static final String ACTION_FIN =
        "net.sgoliver.intent.action.FIN";

    public MiIntentService() {
            super("MiIntentService");
        }

    @Override
    protected void onHandleIntent(Intent intent)
    {
        int iter = intent.getIntExtra("iteraciones", 0);

        for(int i=1; i<=iter; i++) {
            tareaLarga();

            //Comunicamos el progreso
            Intent bcIntent = new Intent();
            bcIntent.setAction(ACTION_PROGRESO);
            bcIntent.putExtra("progreso", i*10);
            sendBroadcast(bcIntent);
        }

        Intent bcIntent = new Intent();
        bcIntent.setAction(ACTION_FIN);
        sendBroadcast(bcIntent);
    }
```

```
    private void tareaLarga()
        {
            try {
                Thread.sleep(1000);
            } catch(InterruptedException e) {}
        }
}
```

Como podéis comprobar también he añadido un nuevo tipo de mensaje broadcast (ACTION_FIN), esta vez sin datos adicionales, para comunicar a la aplicación principal la finalización de la tarea en segundo plano.

Además de la implementación del servicio, recordemos que también tendremos que declararlo en el AndroidManifest.xml, dentro de la sección <application>:

```
<service android:name=".MiIntentService"></service>
```

Y con esto ya tendríamos implementado nuestro servicio. El siguiente paso será llamar al servicio para comenzar su ejecución. Esto lo haremos desde una actividad principal de ejemplo en la que tan sólo colocaremos una barra de progreso y un botón para lanzar el servicio. El código del botón para ejecutar el servicio será muy sencillo, tan sólo tendremos que crear un nuevo intent asociado a la clase MiIntentService, añadir los datos de entrada necesarios mediante putExtra() y ejecutar el servicio llamando a startService() pasando como parámetro el intent de entrada. Como ya dijimos, el único dato de entrada que pasaremos será el número de iteraciones a ejecutar.

```
btnEjecutar.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View v) {
        Intent msgIntent =
                    new Intent(MainActivity.this, MiIntentService.class);
        msgIntent.putExtra("iteraciones", 10);
        startService(msgIntent);
    }
});
```

Con esto ya podríamos ejecutar nuestra aplicación y lanzar la tarea, pero no podríamos ver el progreso de ésta ni saber cuándo ha terminado porque aún no hemos creado el BroadcastReceiver necesario para capturar los mensajes broadcast que envía el servicio durante su ejecución.

Para ello, como clase interna a nuestra actividad principal definiremos una nueva clase que extienda a BroadcastReceiver y que implemente su método onReceive() para gestionar los mensajes ACTION_PROGRESO y ACTION_FIN que definimos en nuestro IntentService. En el caso de recibirse ACTION_PROGRESO extraeremos el nivel de progreso del intent recibido y actualizaremos consecuentemente la barra de progreso mediante setProgress(). En caso de recibirse ACTION_FIN mostraremos un mensaje Toast informando de la finalización de la tarea.

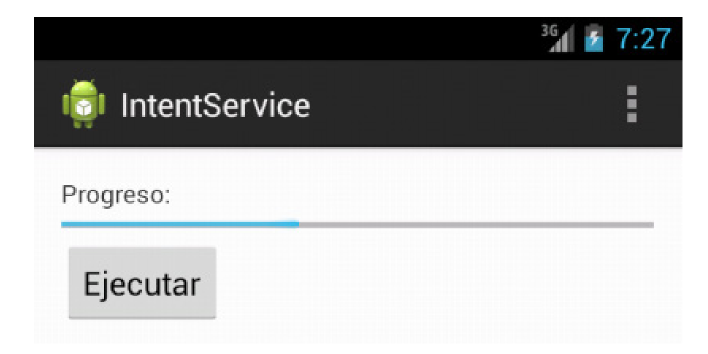
```
public class ProgressReceiver extends BroadcastReceiver {

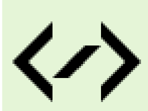
    @Override
    public void onReceive(Context context, Intent intent) {
        if(intent.getAction().equals(MiIntentService.ACTION_PROGRESO)) {
            int prog = intent.getIntExtra("progreso", 0);
            pbarProgreso.setProgress(prog);
        }
        else if(intent.getAction().equals(MiIntentService.ACTION_FIN)) {
            Toast.makeText(MainActivity.this, "Tarea finalizada!", Toast.
LENGTH_SHORT).show();
        }
    }
}
```

Pero aún no habríamos terminado dado, ya que aunque hayamos implementado nuestro `BroadcastReceiver`, éste no tendrá ningún efecto a menos que lo registremos con la aplicación y lo asociemos a los tipos de mensaje que deberá tratar (mediante un `IntentFilter`). Para hacer esto, al final del método `onCreate()` de nuestra actividad principal crearemos un `IntentFilter` al que asociaremos mediante `addAction()` los dos tipos de mensaje broadcast que queremos capturar, instanciaremos nuestro `BroadcastReceiver` y lo registraremos mediante `registerReceiver()`, al que pasaremos la instancia creada y el filtro de mensajes.

```
IntentFilter filter = new IntentFilter();
filter.addAction(MiIntentService.ACTION_PROGRESO);
filter.addAction(MiIntentService.ACTION_FIN);
ProgressReceiver rcv = new ProgressReceiver();
registerReceiver(rcv, filter);
```

Y con esto sí habríamos concluido nuestra aplicación de ejemplo. Si ejecutamos la aplicación en el emulador y pulsamos el botón de comenzar la tarea veremos cómo la barra de progreso comienza a avanzar hasta el final, momento en el que deberá aparecer el mensaje toast indicando la finalización de la tarea.





Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-intentservice

14

Servicios Web

XIV. Acceso a Servicios Web

Servicios Web SOAP: Servidor

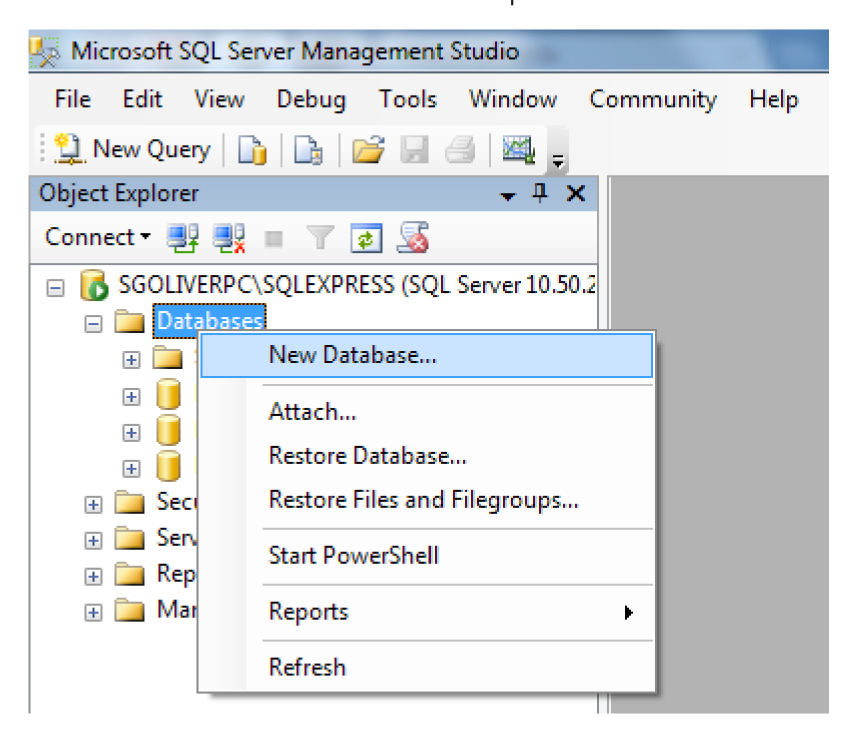
En este primer apartado que vamos a dedicar a los *servicios web* nos vamos a centrar en los servicios que utilizan el estándar **SOAP** como mecanismo de comunicación.

A diferencia de otros tutoriales, no sólo vamos describir cómo acceder a este tipo de servicios desde una aplicación Android, sino que también veremos como crear un servicio web SOAP mediante ASP.NET para acceder a una base de datos SQL Server. De esta forma pretendo ilustrar la "arquitectura" completa de una aplicación Android que acceda a datos almacenados en un servidor de base de datos externo. **Nota**: Aunque intentaré aportar el máximo número de detalles, es imposible abarcarlo todo en un solo capítulo, por lo que el texto supondrá unos conocimientos mínimos de Visual Studio y del lenguaje C#.

Como caso de ejemplo, vamos a crear una aplicación sencilla capaz de gestionar un listado de "clientes" que contendrá el nombre y teléfono de cada uno de ellos. Nuestra aplicación será capaz de consultar el listado actual de clientes almacenados en el servidor externo y de insertar nuevos clientes en la base de datos. Como siempre, se trata de un ejemplo muy sencillo pero creo que lo suficientemente completo como para que sirva de base para crear otras aplicaciones más complejas.

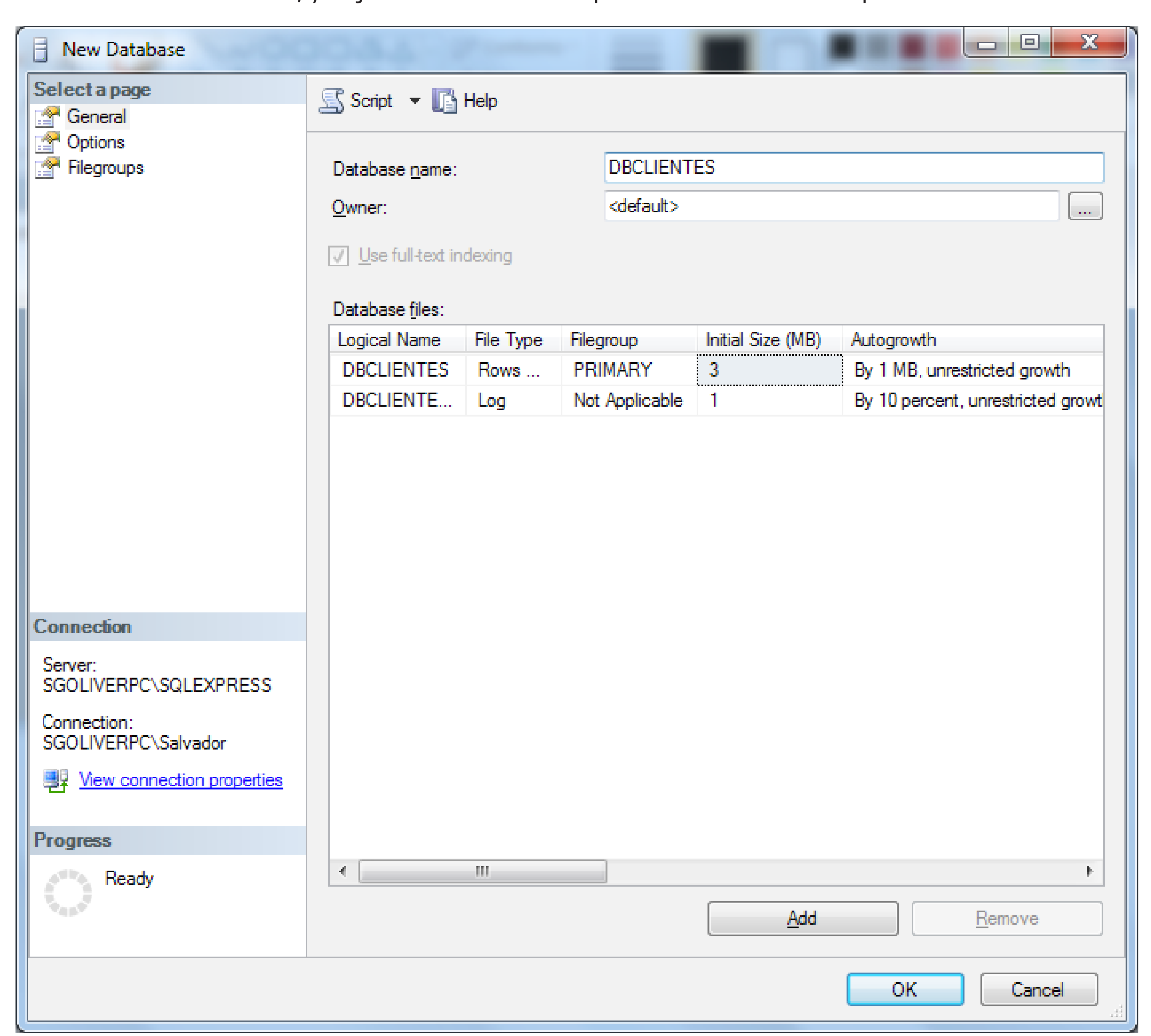
Como software necesario, en este caso utilizaré *Visual Studio 2010* y *SQL Server 2008 R2* para crear el servicio web y la base de datos respectivamente. Podéis descargar de forma gratuita las versiones Express de ambos productos (más que suficientes para crear una aplicación como la que describiremos en este capítulo) desde la web oficial de Microsoft. También es recomendable instalar *SQL Server 2008 Management Studio Express*, descargable también de forma gratuita desde esta web. Esta aplicación no es más que un gestor gráfico para acceder y manipular nuestras bases de datos SQL Server con total comodidad.

Vamos comenzar por la base de todo el sistema, y esto es la base de datos a la que accederá el servicio web y, a través de éste, también la aplicación Android que crearemos más adelante. Para ello abrimos *SQL Server Management Studio*, nos conectamos a nuestro servidor SQL Server local, y pulsamos sobre la sección "Databases" del árbol de objetos que aparece a la izquierda. Sobre esta carpeta podemos acceder a la opción "New Database..." del menú contextual para crear una nueva base de datos.

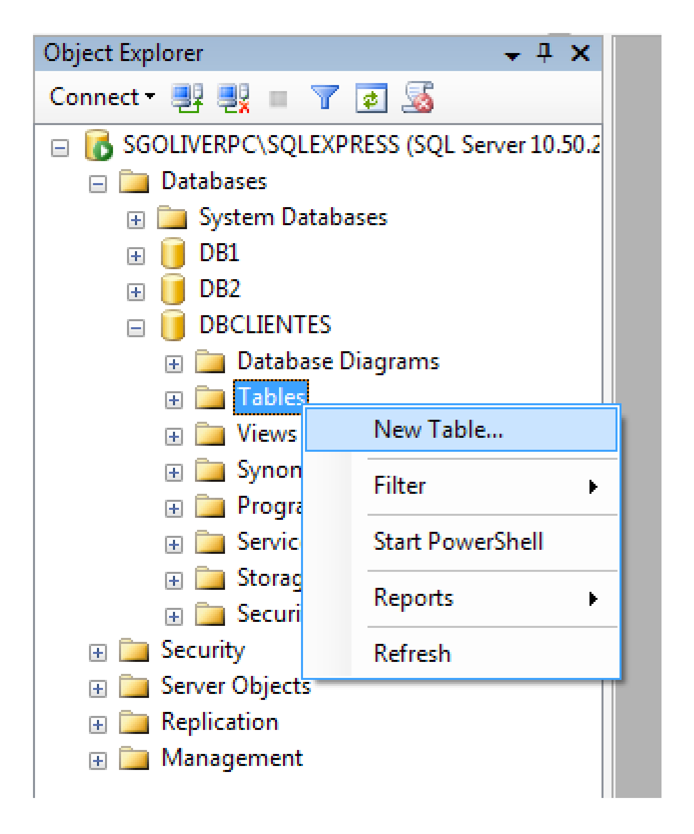


En el cuadro de diálogo que aparece tan sólo indicaremos el nombre de la nueva base de datos, en mi caso

la llamaré DBCLIENTES, y dejaremos el resto de opciones con sus valores por defecto.



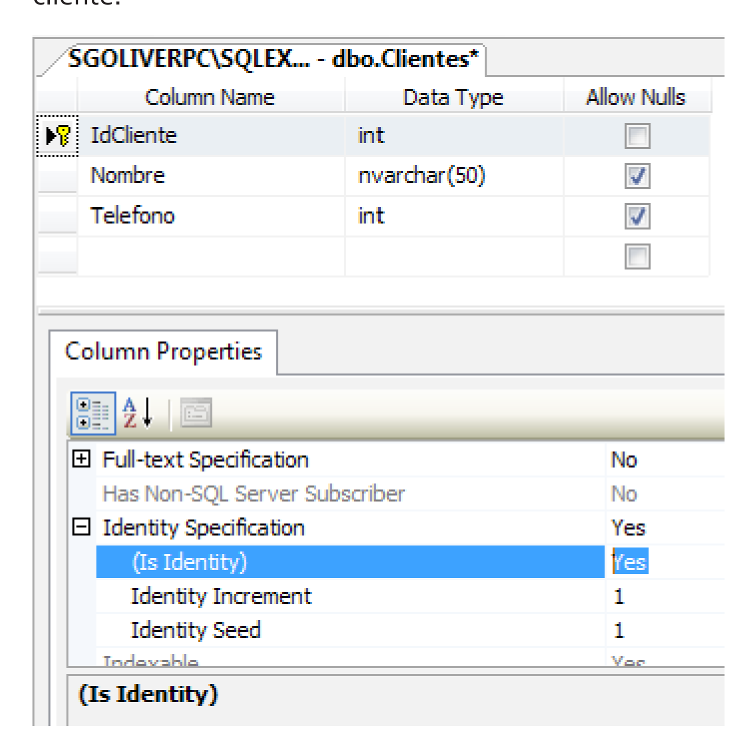
Desplegamos el árbol de carpetas de nuestra recién creada base de datos DBCLIENTES y sobre la carpeta "Tables" ejecutamos la opción "New table..." para crear una nueva tabla.



Vamos a añadir sólo 3 campos a la tabla:

- `IdCliente`, de tipo `int`, que será un código único identificativo del cliente.
- `Nombre`, de tipo `nvarchar(50)`, que contendrá el nombre del cliente.
- `Telefono`, de tipo `int`, que contendrá el teléfono del cliente.

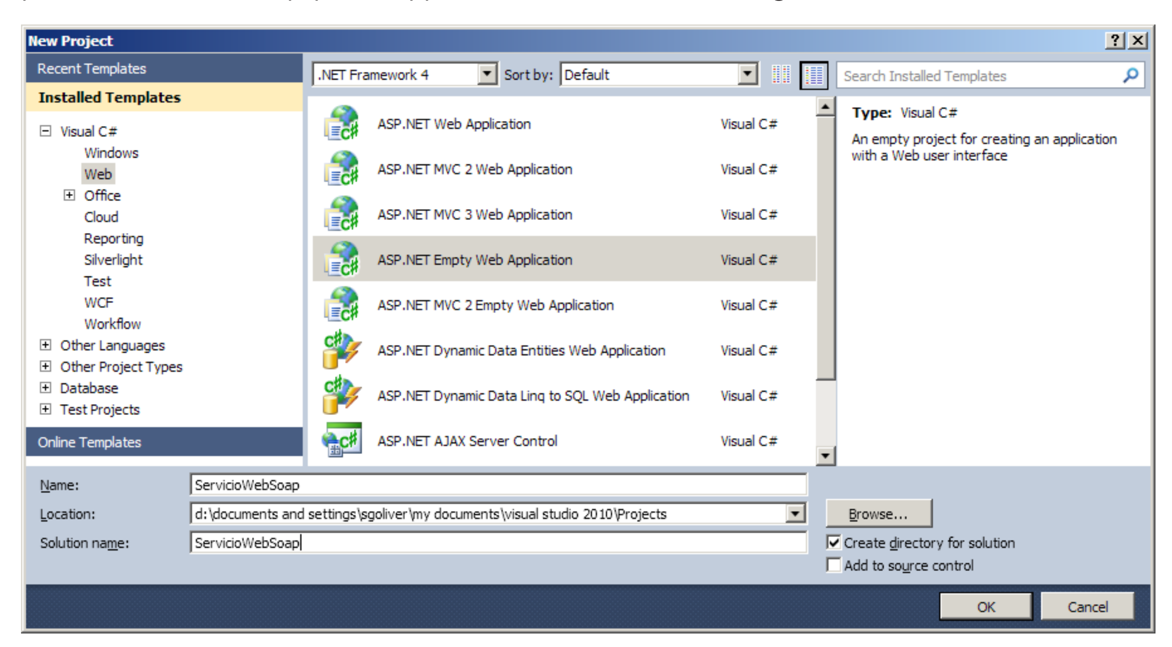
Marcaremos además el campo `IdCliente` como clave principal de la tabla, y también como campo de identidad autoincremental, de modo que se calcule automáticamente cada vez que insertemos un nuevo cliente.



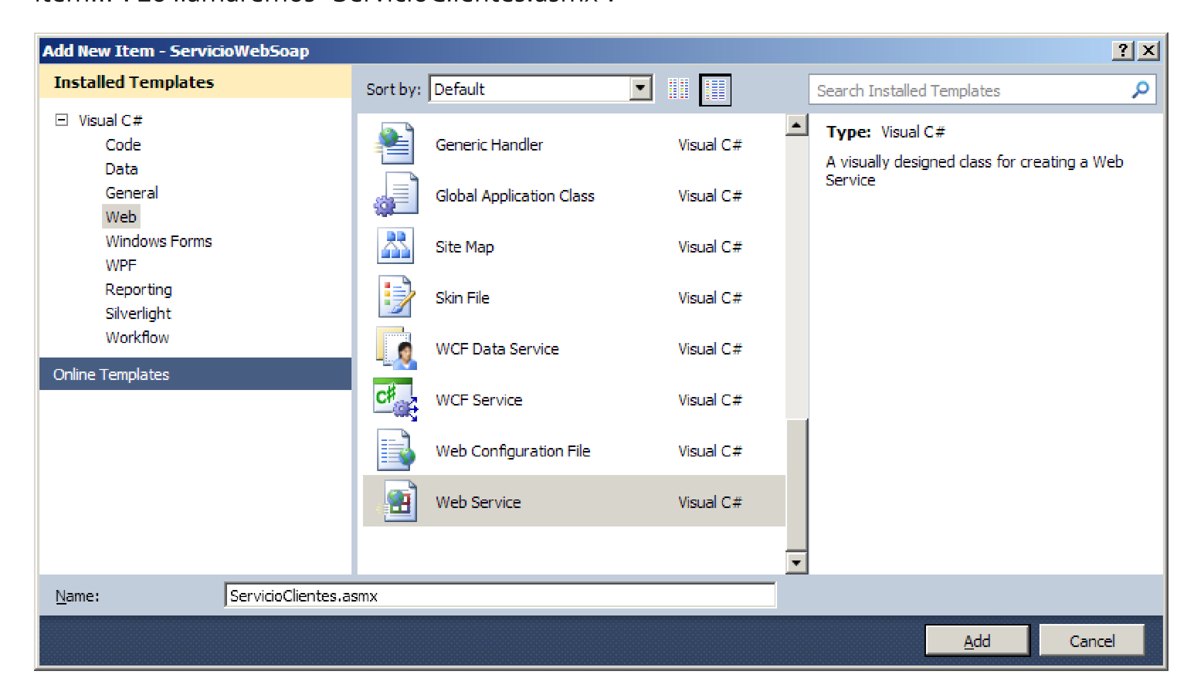
Con esto ya tenemos nuestra tabla finalizada, por lo que sólo nos queda guardarla con el nombre que deseemos, que para este ejemplo será "Clientes".

Hecho, ya tenemos nuestra base de datos SQL Server creada y una tabla preparada para almacenar los datos asociados a nuestros clientes. El siguiente paso será crear el servicio web que manipulará los datos de esta tabla.

Para crear el servicio abriremos Visual Studio 2010 y crearemos un nuevo proyecto web en C# utilizando la plantilla "ASP.NET Empty Web Application". En un alarde de originalidad lo llamaremos "ServicioWebSoap".



Una vez creado el proyecto, añadiremos a éste un nuevo servicio web mediante el menú "Project / Add new item…". Lo llamaremos "ServicioClientes.asmx".



Una vez añadido aparecerá en pantalla el código fuente por defecto del nuevo servicio web, que contiene un único método de ejemplo llamado `HelloWorld()`. Este método podemos eliminarlo ya que no nos servirá de nada, y además modificaremos el atributo `WebService` de la clase para indicar que el namespace será "`http://sgoliver.net/`" (en vuestro caso podéis indicar otro valor). Con esto, nos quedaría un código base como éste:

```
namespace ServicioWebSoap
{
    /// <summary>
    /// Summary description for ServicioClientes
    /// </summary>
    [WebService(Namespace = "http://sgoliver.net/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [System.ComponentModel.ToolboxItem(false)]

    public class ServicioClientes : System.Web.Services.WebService
    {
        //Métodos del servicio web
        //...
    }
}
```

Pues bien, dentro de esta clase `ServicioClientes` es donde añadiremos todos los métodos públicos que queramos tener accesibles a través de nuestro servicio web, siempre precedidos por el atributo `[WebMethod]` como veremos en breve. Para nuestro ejemplo vamos a crear tres métodos, el primero para obtener el listado completo de clientes almacenados en la base de datos, y los otros dos para insertar nuevos clientes (más adelante explicaré por qué dos, aunque adelanto que es tan sólo por motivos didácticos).

Antes de crear estos métodos, vamos a crear una nueva clase sencilla que nos sirva para encapsular los datos de un cliente. La añadiremos mediante la opción "Project / Add class..." de Visual Studio y la llamaremos "Cliente.cs". Esta clase contendrá únicamente los 3 campos que ya comentamos al crear la base de datos y dos constructores, uno de ellos por defecto que tan solo inicializará los campos y otro con parámetros para crear clientes a partir de sus datos identificativos. El código de la clase es muy sencillo, y tan solo cabe mencionar que definiremos sus tres atributos como propiedades automáticas de C# utilizando para ello la notación abreviada `{get; set;}`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace ServicioWebSoap {
    public class Cliente
    {
        public int Id {get; set;}
        public string Nombre {get; set;}
        public int Telefono {get; set;}

        public Cliente()
        {
            this.Id = 0;
            this.Nombre = "";
            this.Telefono = 0;
        }

        public Cliente(int id, string nombre, int telefono)
        {
            this.Id = id;
            this.Nombre = nombre;
            this.Telefono = telefono;
        }
    }
}
```

Vamos ahora a escribir el primero de los métodos que haremos accesible a través de nuestro servicio web. Lo llamaremos `NuevoCliente()`, recibirá como parámetros de entrada un nombre y un teléfono, y se encargará de insertar un nuevo registro en nuestra tabla de clientes con dichos datos. Recordemos que el ID del cliente no será necesario insertarlo de forma explícita ya que lo hemos definido en la base de datos como campo autoincremental. Para el trabajo con la base de datos vamos a utilizar la API clásica de *ADO.NET*, aunque podríamos utilizar cualquier otro mécanismo de acceso a datos, como por ejemplo *Entity Framework*, *NHibernate*, etc.

De esta forma, el primer paso será crear una conexión a SQL Server mediante la clase `SQLConnection`, pasando como parámetro la cadena de conexión correspondiente (en vuestro caso tendréis que modificarla para adaptarla a vuestro entorno). Tras esto abriremos la conexión mediante una llamada al método `Open()`, definiremos el comando SQL que queremos ejecutar creando un objeto `SQLCommand`. Ejecutaremos el comando llamando al método `ExecuteNonQuery()` recogiendo el resultado en una variable, y finalmente cerraremos la conexión llamando a `Close()`. Por último devolveremos el resultado del comando SQL como valor de retorno del método web.

Como podéis ver en el código siguiente, los valores a insertar en la base de datos los hemos especificado en la consulta SQL como parámetros variable (precedidos por el carácter '`@`'). Los valores de estos parámetros los definimos y añadimos al comando SQL mediante el método `Add()` de su propiedad `Parameters`. Esta opción es más recomendable que la opción clásica de concatenar directamente la cadena de texto de la sentencia SQL con los parámetros variables, ya que entre otras cosas servirá para evitar [en gran medida] posibles ataques de inyección SQL. El resultado devuelto por este método será el número de registros afectados por la sentencia SQL ejecutada, por lo que para verificar si se ha ejecutado correctamente bastará con comprobar que el resultado es igual a 1.

```
[WebMethod]
public int NuevoClienteSimple(string nombre, int telefono)
{
    SqlConnection con =
        new SqlConnection(
            @"Data Source=SGOLIVERPC\SQLEXPRESS;Initial
Catalog=DBCLIENTES;Integrated Security=True");

    con.Open();

    string sql =
       "INSERT INTO Clientes (Nombre, Telefono) VALUES (@nombre, @telefono)";

    SqlCommand cmd = new SqlCommand(sql, con);

    cmd.Parameters
           .Add("@nombre", System.Data.SqlDbType.NVarChar).Value = nombre;

    cmd.Parameters
           .Add("@telefono", System.Data.SqlDbType.Int).Value = telefono;

    int res = cmd.ExecuteNonQuery();

    con.Close();

    return res;
}
```

En el código anterior, podéis ver que hemos precedido el método con el atributo `[WebMethod]`. Con este atributo estamos indicando que el método será accesible a través de nuestro servicio web y podrá ser llamado desde cualquier aplicación que se conecte con éste.

La siguiente operación que vamos a añadir a nuestro servicio web será la que nos permita obtener el listado

completo de clientes registrados en la base de datos. Llamaremos al método `ListadoClientes()` y devolverá un array de objetos de tipo `Cliente`. El código del método será muy similar al ya comentado para la operación de inserción, con la única diferencia de que en esta ocasión la sentencia SQL será obviamente un SELECT y que utilizaremos un objeto `SqlDataReader` para leer los resultados devueltos por la consulta. Los registros leídos los iremos añadiendo a una lista de tipo `List<Clientes>` y una vez completada la lectura convertiremos esta lista en un array de clientes llamando al método `ToArray()`. Este último array será el que devolveremos como resultado del método. Veamos el código completo del método:

```
[WebMethod]
public Cliente[] ListadoClientes()
{
    SqlConnection con =
        new SqlConnection(
            @"Data Source=SGOLIVERPC\SQLEXPRESS;Initial
Catalog=DBCLIENTES;Integrated Security=True");

    con.Open();

    string sql = "SELECT IdCliente, Nombre, Telefono FROM Clientes";

    SqlCommand cmd = new SqlCommand(sql, con);

    SqlDataReader reader = cmd.ExecuteReader();

    List<Cliente> lista = new List<Cliente>();

    while (reader.Read())
    {
        lista.Add(
            new Cliente(reader.GetInt32(0),
                        reader.GetString(1),
                        reader.GetInt32(2)));
    }

    con.Close();

    return lista.ToArray();
}
```

Por último, como dijimos al principio, vamos a añadir un tercer método web con fines puramente didácticos. Si os fijáis en los dos métodos anteriores, veréis que en uno de los casos devolvemos como resultado un valor simple, un número entero, y en el otro caso un objeto complejo, en concreto un array de objetos de tipo `Cliente`. Sin embargo, ninguno de ellos recibe como parámetro un tipo complejo, tan sólo valores simples (enteros y strings). Esto no tiene mucha relevancia en el código de nuestro servicio web, pero sí tiene ciertas peculiaridades a la hora de realizar la llamada al servicio desde la aplicación Android. Por lo que para poder explicar esto más adelante añadiremos un nuevo método de inserción de clientes que, en vez de recibir los parámetros de nombre y teléfono por separado, recibirá como dato de entrada un objeto `Cliente`.

El código de este método, que llamaremos `NuevoClienteObjeto()`, será exactamente igual al anterior método de inserción, con la única diferencia de los parámetros de entrada, por lo que no nos detendremos en comentar nada más.

```
[WebMethod]
public int NuevoClienteObjeto(Cliente cliente)
{
    SqlConnection con = new SqlConnection(@"Data Source=SGOLIVERPC\
SQLEXPRESS;Initial Catalog=DBCLIENTES;Integrated Security=True");

    con.Open();

    string sql =
        "INSERT INTO Clientes (Nombre, Telefono) VALUES (@nombre, @telefono)";

    SqlCommand cmd = new SqlCommand(sql, con);

    cmd.Parameters
       .Add("@nombre", System.Data.SqlDbType.NVarChar).Value = cliente.Nombre;

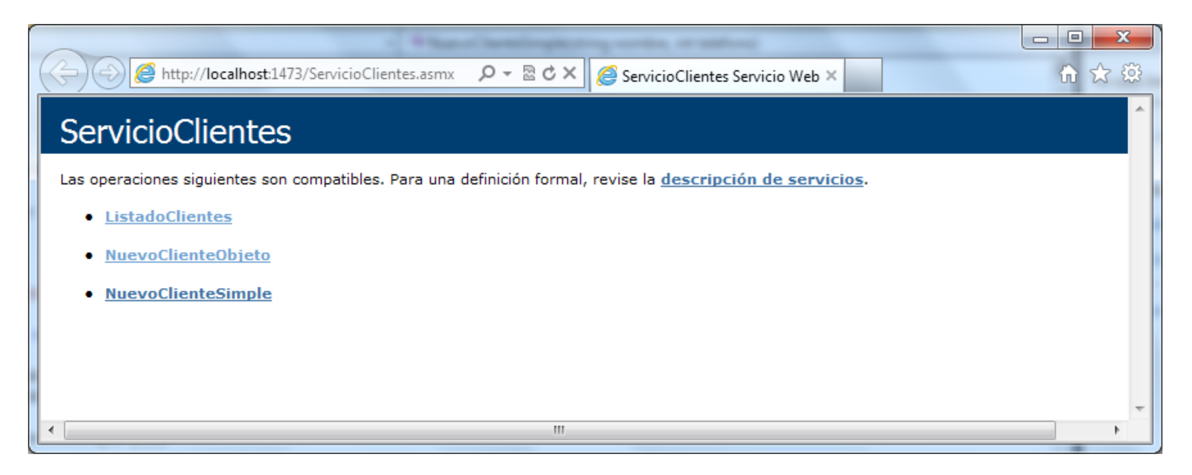
    cmd.Parameters
       .Add("@telefono", System.Data.SqlDbType.Int).Value = cliente.Telefono;

    int res = cmd.ExecuteNonQuery();

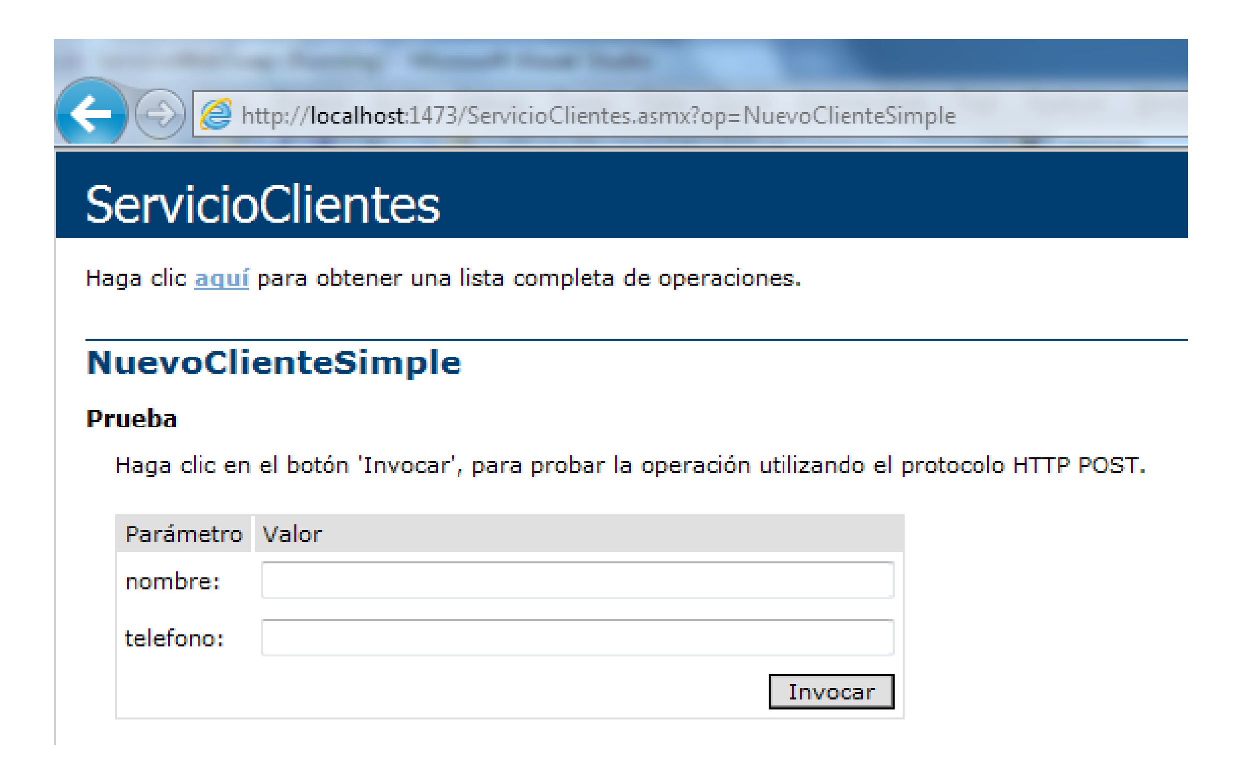
    con.Close();

    return res;
}
```

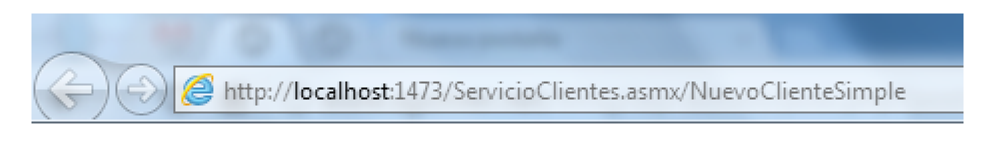
Y con esto hemos finalizado nuestro servicio web. Podemos probar su funcionamiento con la página de prueba que proporciona ASP.NET al ejecutar el proyecto en Visual Studio. Si ejecutamos el proyecto se abrirá automáticamente un explorador web que mostrará una página con todas las operaciones que hemos definido en el servicio web.



Si pulsamos sobre cualquiera de ellas pasaremos a una nueva página que nos permitirá dar valores a sus parámetros y ejecutar el método correspondiente para visualizar sus resultados. Si pulsamos por ejemplo en la operación `NuevoCliente(string, int)` llegaremos a esta página:



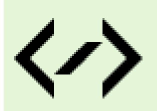
Aquí podemos dar valores a los dos parámetros y ejecutar el método (botón "*Invoke*"), lo que nos devolverá la respuesta codificada en un XML según el estandar SOAP.




```
<?xml version="1.0" encoding="UTF-8"?>
<int xmlns="http://sgoliver.net/">1</int>
```

Como podéis comprobar, en principio el XML devuelto no es fácil de interpretar, pero esto es algo que no debe preocuparnos demasiado ya que en principio será transparente para nosotros, las librerías que utilizaremos más adelante en Android para la llamada a servicios SOAP se encargarán de *parsear* convenientemente estas respuestas y de darnos tan sólo aquella parte que necesitamos.

En el siguiente apartado nos ocuparemos de la construcción de una aplicación Android que sea capaz de conectarse a este servicio web y de llamar a los métodos que hemos definido para insertar y recuperar clientes de nuestra base de datos. Veremos además cómo podemos ejecutar y probar en local todo el sistema de forma que podamos comprobar que todo funciona como esperamos.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/dotnet-serviciowebsoap

Servicios Web SOAP: Cliente

En el apartado anterior vimos cómo construir un servicio web SOAP haciendo uso de ASP.NET y una base de datos externa SQL Server. En este segundo apartado sobre SOAP veremos cómo podemos acceder a este servicio web desde una aplicación Android y probaremos todo el sistema en local para verificar su correcto

funcionamiento.

En primer lugar hay que empezar diciendo que Android no incluye "de serie" ningún tipo de soporte para el acceso a servicios web de tipo SOAP. Es por esto por lo que vamos a utilizar una librería externa para hacernos más fácil esta tarea. Entre la oferta actual, la opción más popular y más utilizada es la librería ksoap2-android. Esta librería es un fork, especialmente adaptado para Android, de la antigua librería kSOAP2. Este framework nos permitirá de forma relativamente fácil y cómoda utilizar servicios web que utilicen el estándar SOAP. La última versión de esta librería en el momento de escribir este texto es la 3.0.0, que puede descargarse desde este enlace.

Agregar esta librería a nuestro proyecto Android es muy sencillo. Si tenemos una versión reciente del plugin de Android para Eclipse, una vez tenemos creado el proyecto en Android bastará con copiar el archivo `.jar` en la carpeta `libs` de nuestro proyecto. Si tu versión del plugin es más antigua es posible que tengas que además añadir la librería al *path* del proyecto. Para ello accederemos al menú "Project / Properties" y en la ventana de propiedades accederemos a la sección "Java Build Path". En esta sección accederemos a la solapa "Libraries" y pulsaremos el botón "Add External JARs...". Aquí seleccionamos el fichero jar de la librería ksoap2-android (en este caso "ksoap2-android-assembly-3.6.0-jar-with-dependencies.jar") y listo, ya tenemos nuestro proyecto preparado para hacer uso de la funcionalidad aportada por la librería.

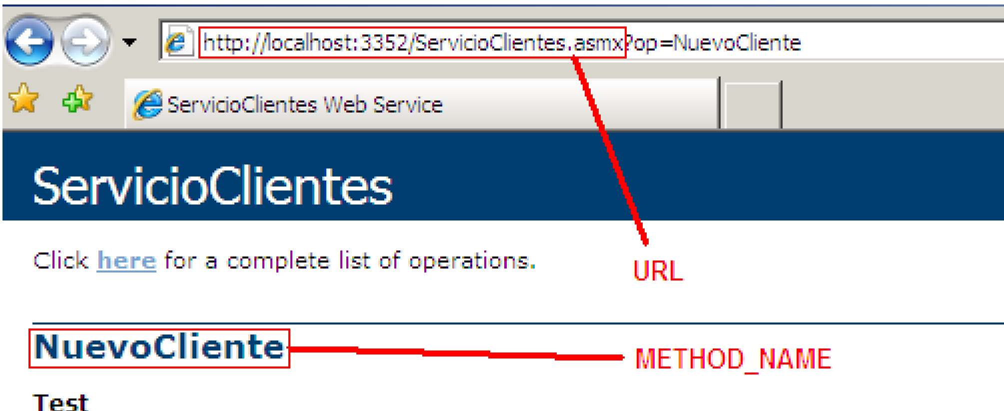
Como aplicación de ejemplo, vamos a crear una aplicación sencilla que permita añadir un nuevo usuario a la base de datos. Para ello añadiremos a la vista principal dos cuadros de texto para introducir el nombre y teléfono del nuevo cliente (en mi caso se llamarán `txtNombre` y `txtTelefono` respectivamente) y un botón (en mi caso `btnEnviar`) que realice la llamada al método `NuevoCliente` del servicio web pasándole como parámetros los datos introducidos en los cuadros de texto anteriores.

No voy a mostrar todo el código necesario para crear esta vista y obtener las referencias a cada control porque no tiene ninguna particularidad sobre lo ya visto en multitud de ocasiones en capítulos anteriores del libro. Lo que nos interesa en este caso es la implementación del evento `onClick` del botón `btnEnviar`, que será el encargado de comunicarse con el servicio web y procesar el resultado.

Lo primero que vamos a hacer en este evento es definir, por comodidad, cuatro constantes que nos servirán en varias ocasiones durante el código:

- `NAMESPACE`. Espacio de nombres utilizado en nuestro servicio web.
- `URL`. Dirección URL para realizar la conexión con el servicio web.
- `METHOD_NAME`. Nombre del método web concreto que vamos a ejecutar.
- `SOAP_ACTION`. Equivalente al anterior, pero en la notación definida por SOAP.

Aunque los valores se podrían más o menos intuir, para conocer exactamente los valores que debemos asignar a estas constantes vamos a ejecutar una vez más el proyecto de Visual Studio que construimos en el apartado anterior y vamos a acceder a la página de prueba del método `NuevoCliente`. Veremos algo parecido a lo siguiente:



Test

To test the operation using the HTTP POST protocol, click the 'Invoke' button.

Parameter	Value
nombre:	
telefono:	

Invoke

SOAP 1.1

The following is a sample SOAP 1.1 request and response. The **placeholders** sho

```
POST /ServicioClientes.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://sgoliver.net/NuevoCliente"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  <soap:Body>
    <NuevoCliente xmlns="http://sgoliver.net/">
      <nombre>string</nombre>
      <telefono>int</telefono>
    </NuevoCliente>
  </soap:Body>
</soap:Envelope>
```

SOAP_ACTION

NAMESPACE

En la imagen anterior se muestran resaltados en rojo los valores de las cuatro constantes a definir, que en nuestro caso concreto quedarían de la siguiente forma:

```
String NAMESPACE = "http://sgoliver.net/";
String URL="http://10.0.2.2:1473/ServicioClientes.asmx";
String METHOD_NAME = "NuevoClienteSimple";
String SOAP_ACTION = "http://sgoliver.net/NuevoClienteSimple";
```

Como podéis comprobar, y esto es algo importante, en la URL he sustituido el nombre de máquina `localhost` por su dirección IP equivalente, que en el caso de aplicaciones Android ejecutadas en el emulador se corresponde con la dirección `10.0.2.2`, en vez de la clásica `127.0.0.1`. Además debes verificar que el puerto coincide con el que estás utilizando en tu máquina. En mi caso el servicio se ejecuta sobre el puerto `1473`, pero es posible que en tu caso el número sea distinto.

Los siguientes pasos del proceso serán crear la petición SOAP al servicio web, enviarla al servidor y recibir

la respuesta. Aunque ya dijimos que todo este proceso sería casi transparente para el programador, por ser ésta la primera vez que hablamos del tema me voy a detener un poco más para intentar que entendamos lo que estamos haciendo y no solo nos limitemos a copiar/pegar trozos de código que no sabemos lo que hacen.

Volvamos a la página de prueba del método web `NuevoCliente`. Justo debajo de la sección donde se solicitan los parámetros a pasar al método se incluye también un XML de muestra de cómo tendría que ser nuestra petición al servidor si tuviéramos que construirla a mano. Echémosle un vistazo:

SOAP 1.1

The following is a sample SOAP 1.1 request and response. The **placeholders** shown need to be replaced with actual values.

```
POST /ServicioClientes.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://sgoliver.net/NuevoCliente"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www
  <soap:Body>
    <NuevoCliente xmlns="http://sgoliver.net/">
      <nombre>string</nombre>
      <telefono>int</telefono>
    </NuevoCliente>
  </soap:Body>
</soap:Envelope>
```

Encabezados HTTP

REQUEST

ENVELOPE

Una vez más he marcado varias zonas sobre la imagen, correspondientes a las tres partes principales de una petición de tipo SOAP. Empezando por la "parte interna" del XML, en primer lugar encontramos los datos de la petición en sí (*Request*) que contiene el nombre del método al que queremos llamar, y los nombres y valores de los parámetros en entrada. Rodeando a esta información se añaden otra serie de etiquetas y datos a modo de contenedor estándar que suele recibir el nombre de *Envelope*. La información indicada en este contenedor no es específica de nuestra llamada al servicio, pero sí contiene información sobre formatos y esquemas de validación del estándar SOAP. Por último, durante el envío de esta petición SOAP al servidor mediante el protocolo HTTP se añaden determinados encabezados como los que veis en la imagen. Todo esto junto hará que el servidor sea capaz de interpretar correctamente nuestra petición SOAP, se llame al método web correcto, y se devuelva el resultado en un formato similar al anterior que ya veremos más adelante. Aclarada un poco la estructura y funcionamiento general de una petición SOAP veamos lo sencillo que resulta realizarla desde nuestra aplicación Android.

En primer lugar crearemos la petición (*request*) a nuestro método `NuevoCliente`. Para ello crearemos un nuevo objeto `SoapObject` pasándole el *namespace* y el nombre del método web. A esta petición tendremos que asociar los parámetros de entrada mediante el método `addProperty()` al que pasaremos los nombres y valores de los parámetros (que en nuestro caso se obtendrán de los cuadros de texto de la vista principal).

```
SoapObject request = new SoapObject(NAMESPACE, METHOD_NAME);

request.addProperty("nombre", txtNombre.getText().toString());
request.addProperty("telefono", txtTelefono.getText().toString());
```

El segundo paso será crear el contenedor SOAP (*envelope*) y asociarle nuestra petición. Para ello crearemos un nuevo objeto `SoapSerializationEnvelope` indicando la versión de SOAP que vamos a usar (versión 1.1 en nuestro caso, como puede verse en la imagen anterior). Indicaremos además que se trata de un servicio web .NET activando su propiedad dotNet. Por último, asociaremos la petición antes creada a nuestro contenedor llamando al método `setOutputSoapObject()`.

```
SoapSerializationEnvelope envelope =
    new SoapSerializationEnvelope(SoapEnvelope.VER11);

envelope.dotNet = true;

envelope.setOutputSoapObject(request);
```

Como tercer paso crearemos el objeto que se encargará de realizar la comunicación HTTP con el servidor, de tipo `HttpTransportSE`, al que pasaremos la URL de conexión a nuestro servicio web. Por último, completaremos el proceso realizando la llamada al servicio web mediante el método `call()`.

```
HttpTransportSE transporte = new HttpTransportSE(URL);

try
{
    transporte.call(SOAP_ACTION, envelope);

    //Se procesa el resultado devuelto
    //...
}
catch (Exception e)
{
    txtResultado.setText("Error!");
}
```

Tras la llamada al servicio ya estamos en disposición de obtener el resultado devuelto por el método web llamado. Esto lo conseguimos mediante el método `getResponse()`. Dependiendo del tipo de resultado que esperemos recibir deberemos convertir esta respuesta a un tipo u otro. En este caso, como el resultado que esperamos es un valor simple (un número entero) convertiremos la respuesta a un objeto `SoapPrimitive`, que directamente podremos convertir a una cadena de caracteres llamado a `toString()`. Más adelante veremos cómo tratar valores de retorno más complejos.

```
SoapPrimitive resultado_xml =(SoapPrimitive)envelope.getResponse();
String res = resultado_xml.toString();

if(res.equals("1"))
    txtResultado.setText("Insertado OK");
```

Y listo, con esto ya tenemos preparada la llamada a nuestro servicio web y el tratamiento de la respuesta recibida.

Un detalle más antes de poder probar todo el sistema. Debemos acordarnos de conceder permiso de acceso a internet a nuestra aplicación, añadiendo la linea correspondiente al *Android Manifest*:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Pues bien, para probar lo que llevamos hasta ahora podemos ejecutar ambos proyectos simultáneamente, en primer lugar el de Visual Studio para iniciar la ejecución del servidor local que alberga nuestro servicio web (hay que dejar abierto el explorador una vez que se abra), y posteriormente el de Eclipse para iniciar nuestra aplicación Android en el Emulador. Una vez están los dos proyectos en ejecución, podemos rellenar los datos de nuestro cliente en la aplicación Android y pulsar el botón "Enviar" para realizar la llamada al servicio web e insertar el cliente en la base de datos (que por supuesto también deberá estar iniciada). Si todo va bien y no se produce ningún error, deberíamos poder consultar la tabla de Clientes a través del SQL Server Management Studio para verificar que el cliente se ha insertado correctamente.



En la imagen vemos cómo hemos insertado un nuevo cliente llamada 'cliente7' con número de teléfono '7777'. Si consultamos ahora nuestra base de datos Sql Server podremos comprobar si el registro efectivamente se ha insertado correctamente.

Results | Messages

	IdCliente	Nombre	Telefono
1	1	cliente1	1111
2	2	cliente2	2222
3	3	cliente3	3333
4	4	cliente4	4444
5	5	cliente5	5555
6	6	cliente6	6666
7	7	cliente7	7777

Algo importante que quiero remarcar llegados a este punto. El código anterior debe funcionar correctamente sobre un dispositivo o emulador con versión de Android anterior a la 3.0. Sin embargo, si intentamos ejecutar la aplicación sobre una versión posterior obtendremos una excepción de tipo `NetworkOnMainThread`. Esto es debido a que en versiones recientes de Android no se permite realizar operaciones de larga duración directamente en el hilo principal de la aplicación. Para solucionar esto y que nuestra aplicación funcione bajo cualquier versión de Android será necesario trasladar el código que hemos escrito para llamar al servicio web a una `AsyncTask` que realice las operaciones en segundo plano utilizando un hilo secundario. El curso contiene un capitulo dedicado a describir con más detalle las tareas asíncronas o `AsyncTask`, por lo que en este caso me limitaré a poner cómo quedaría nuestro código dentro de la `AsyncTask`.

```
//Tarea Asíncrona para llamar al WS de consulta en segundo plano
private class TareaWSConsulta extends AsyncTask<String,Integer,Boolean> {

    private Cliente[] listaClientes;

    protected Boolean doInBackground(String... params) {

      boolean resul = true;

      final String NAMESPACE = "http://sgoliver.net/";
      final String URL="http://10.0.2.2:1473/ServicioClientes.asmx";
      final String METHOD_NAME = "ListadoClientes";
      final String SOAP_ACTION = "http://sgoliver.net/ListadoClientes";

      SoapObject request = new SoapObject(NAMESPACE, METHOD_NAME);

      SoapSerializationEnvelope envelope =
           new SoapSerializationEnvelope(SoapEnvelope.VER11);
```

```
        envelope.dotNet = true;

        envelope.setOutputSoapObject(request);

        HttpTransportSE transporte = new HttpTransportSE(URL);

        try
        {
              transporte.call(SOAP_ACTION, envelope);

              SoapObject resSoap =(SoapObject)envelope.getResponse();

              listaClientes = new Cliente[resSoap.getPropertyCount()];

              for (int i = 0; i < listaClientes.length; i++)
              {
                   SoapObject ic = (SoapObject)resSoap.getProperty(i);

                   Cliente cli = new Cliente();
                   cli.id = Integer.parseInt(ic.getProperty(0).toString());
                   cli.nombre = ic.getProperty(1).toString();
                   cli.telefono =
                          Integer.parseInt(ic.getProperty(2).toString());

                   listaClientes[i] = cli;
               }
        }
        catch (Exception e)
        {
              resul = false;
        }

         return resul;
     }

     protected void onPostExecute(Boolean result) {

        if (result)
        {
              //Rellenamos la lista con los nombres de los clientes
              final String[] datos = new String[listaClientes.length];

              for(int i=0; i<listaClientes.length; i++)
                     datos[i] = listaClientes[i].nombre;

              ArrayAdapter<String> adaptador =
                  new ArrayAdapter<String>(MainActivity.this,
                      android.R.layout.simple_list_item_1, datos);

              lstClientes.setAdapter(adaptador);
        }
        else
        {
              txtResultado.setText("Error!");
        }
      }
}
```

Como podemos ver, prácticamente todo el código se ha trasladado al método `doInBackground()` de la tarea, salvo la parte en la que debemos actualizar la interfaz de usuario tras la llamada que debe ir al método `onPostExecute()`.

Por su parte, una vez creada la tarea asíncrona, en el evento click del botón nos limitaremos a instanciar la tarea y ejecutarla llamando a su método `execute()`.

```
btnConsultar.setOnClickListener(new OnClickListener() {

      @Override
      public void onClick(View v) {
            TareaWSConsulta tarea = new TareaWSConsulta();
              tarea.execute();
      }
});
```

Con esto, ya sabemos realizar una llamada a un servicio web SOAP que devuelve un valor de retorno sencillo, en este caso un simple número entero. Lo siguiente que vamos a ver será como implementar la llamada a un método del servicio web que nos devuelva un valor algo más complejo. Y esto lo vamos a ver con la llamada al método web `ListadoClientes()` que recordemos devolvía un array de objetos de tipo `Cliente`.

En este caso, la llamada al método web se realizará de forma totalmente análoga a la ya comentada. Donde llegarán las diferencias será a la hora de tratar el resultado devuelto por el servicio, comenzando por el resultado del método `getResponse()` de ksoap. En esta ocasión, dado que el resultado esperado no es ya un valor simple sino un objeto más complejo, convertiremos el resultado de `getResponse()` al tipo `SoapObject`, en vez de `SoapPrimitive` como hicimos anteriormente. Nuestro objetivo será generar un array de objetos `Cliente` (lo llamaremos `listaClientes`) a partir del resultado devuelto por la llamada al servicio.

Como sabemos que el resultado devuelto por el servicio es también un array, lo primero que haremos será crear un array local con la misma longitud que el devuelto, lo que conseguiremos mediante el método `getPropertyCount()`. Tras esto, iteraremos por los distintos elementos del array devuelto mediante el método `getProperty(ind)`, donde ind será el índice de cada ocurrencia. Cada uno de estos elementos será a su vez otro objeto de tipo `SoapObject`, que representará a un `Cliente`. Adicionalmente, para cada elemento accederemos a sus propiedades (`Id`, `Nombre`, y `Telefono`) una vez más mediante llamadas a `getProperty()`, con el índice de cada atributo, que seguirá el mismo orden en que se definieron. Así, `getProperty(0)` recuperará el Id del cliente, `getProperty(1)` el nombre, y `getProperty(2)` el teléfono. De esta forma podremos crear nuestros objetos `Cliente` locales a partir de estos datos. Al final de cada iteración añadimos el nuevo cliente recuperado a nuestro array. Veamos como quedaría todo esto en el código, donde seguro que se entiende mejor:

```
SoapObject resSoap =(SoapObject)envelope.getResponse();

Cliente[] listaClientes = new Cliente[resSoap.getPropertyCount()];

for (int i = 0; i < listaClientes.length; i++)
{
       SoapObject ic = (SoapObject)resSoap.getProperty(i);

       Cliente cli = new Cliente();
       cli.id = Integer.parseInt(ic.getProperty(0).toString());
       cli.nombre = ic.getProperty(1).toString();
       cli.telefono = Integer.parseInt(ic.getProperty(2).toString());

       listaClientes[i] = cli;
}
```

En nuestra aplicación de ejemplo añadimos un nuevo botón y un control tipo lista (lo llamo `lstClientes`), de forma que al pulsar dicho botón rellenemos la lista con los nombres de todos los clientes recuperados. La forma de rellenar una lista con un array de elementos ya la vimos en los apartados dedicados a los controles de selección, por lo que no nos pararemos a comentarlo. El código sería el siguiente (Nota: sé que todo esto se podría realizar de forma más eficiente sin necesidad de crear distintos arrays para los clientes y para el adaptador de la lista, pero lo dejo así para no complicar el tutorial con temas ya discutidos en otros apartados):

```
//Rellenamos la lista con los nombres de los clientes
final String[] datos = new String[listaClientes.length];

for(int i=0; i<listaClientes.length; i++)
    datos[i] = listaClientes[i].nombre;

ArrayAdapter<String> adaptador =
    new ArrayAdapter<String>(ServicioWebSoap.this,
        android.R.layout.simple_list_item_1, datos);

lstClientes.setAdapter(adaptador);
```

Por último, vamos a ver cómo llamar a un método web que recibe como parámetro algún objeto complejo. Para ilustrarlo haremos una llamada al segundo método de inserción de clientes que implementamos en el servicio, `NuevoClienteObjeto()`. Recordemos que este método recibía como parámetro de entrada un objeto de tipo `Cliente`.

Para poder hacer esto, lo primero que tendremos que hacer será modificar un poco nuestra clase `Cliente`, de forma que ksoap sepa cómo serializar nuestros objetos `Cliente` a la hora de generar las peticiones SOAP correspondientes. Y para esto, lo que haremos será implementar la interfaz `KvmSerializable` en nuestra clase `Cliente`. Para ello, además de añadir la cláusula implements correspondiente tendremos que implementar los siguientes métodos:

- `getProperty(int indice)`.
- `getPropertyCount()`.
- `getPropertyInfo(int indice, HashTable ht, PropertyInfo info)`.
- `setProperty(int indice, Object valor)`.

El primero de ellos deberá devolver el valor de cada atributo de la clase a partir de su índice de orden. Así, para el índice 0 se devolverá el valor del atributo `Id`, para el índice 1 el del atributo `Nombre`, y para el 2 el atributo `Teléfono`.

```
@Override
public Object getProperty(int arg0) {

    switch(arg0) {
        case 0:
            return id;
        case 1:
            return nombre;
        case 2:
            return telefono;
        }

    return null;
}
```

El segundo de los métodos, deberá devolver simplemente el número de atributos de nuestra clase, que en nuestro caso será 3 (`Id`, `Nombre` y `Telefono`):

```
@Override
public int getPropertyCount() {
    return 3;
}
```

El objetivo del tercero será informar, según el índice recibido como parámetro, el tipo y nombre del atributo correspondiente. El tipo de cada atributo se devolverá como un valor de la clase `PropertyInfo`.

```
@Override
public void getPropertyInfo(int ind, Hashtable ht, PropertyInfo info) {
    switch(ind)
        {
        case 0:
            info.type = PropertyInfo.INTEGER_CLASS;
            info.name = "Id";
            break;
        case 1:
            info.type = PropertyInfo.STRING_CLASS;
            info.name = "Nombre";
            break;
        case 2:
            info.type = PropertyInfo.INTEGER_CLASS;
            info.name = "Telefono";
            break;
        default:break;
        }
}
```

Por último, el método `setProperty()` será el encargado de asignar el valor de cada atributo según su índice y el valor recibido como parámetro.

```
@Override
public void setProperty(int ind, Object val) {
    switch(ind)
        {
        case 0:
            id = Integer.parseInt(val.toString());
            break;
        case 1:
            nombre = val.toString();
            break;
        case 2:
            telefono = Integer.parseInt(val.toString());
            break;
        default:
            break;
        }
}
```

Mediante estos métodos, aunque de forma transparente para el programador, ksoap será capaz de transformar nuestros objetos `Cliente` al formato XML correcto de forma que pueda pasarlos como parámetro en las peticiones SOAP a nuestro servicio.

Por su parte, la llamada al servicio también difiere un poco de lo ya comentado a la hora de asociar los

parámetros de entrada del método web. En este caso, construiremos en primer lugar el objeto `Cliente` que queremos insertar en la base de datos a partir de los datos introducidos en la pantalla de nuestra aplicación de ejemplo. Tras esto crearemos un nuevo objeto `PropertyInfo`, al que asociaremos el nombre, valor y tipo de nuestro cliente mediante sus métodos `setName()`, `setValue()` y `setClass()` respectivamente. Por último, asociaremos este cliente como parámetro de entrada al servicio llamando al metodo `addProperty()` igual que hemos hecho en las anteriores ocasiones, con la diferencia de que esta vez lo llamaremos pasándole el objeto `PropertyInfo` que acabamos de crear. Además de esto, tendremos también que llamar finalmente al método `addMapping()` para asociar de alguna forma nuestro espacio de nombres y nombre de clase "Cliente" con la clase real java. Veamos el código para entenderlo mejor:

```
Cliente cli = new Cliente();
cli.nombre = txtNombre.getText().toString();
cli.telefono = Integer.parseInt(txtTelefono.getText().toString());

PropertyInfo pi = new PropertyInfo();
pi.setName("cliente");
pi.setValue(cli);
pi.setType(cli.getClass());

request.addProperty(pi);

SoapSerializationEnvelope envelope =
    new SoapSerializationEnvelope(SoapEnvelope.VER11);
envelope.dotNet = true;

envelope.setOutputSoapObject(request);

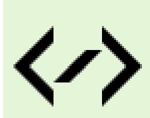
envelope.addMapping(NAMESPACE, "Cliente", cli.getClass());
```

Todo esto lo haremos en un nuevo botón añadido a la aplicación de ejemplo (*Enviar2*), cuyo efecto tendrá que ser idéntico al que ya creamos para la llamada al método web `NuevoClienteSimple()`, aunque como acabamos de ver su implementación es algo diferente debido a los distintos parámetros de entrada utilizados.

Como imagen final veamos una captura de la pantalla final de nuestra aplicación de ejemplo, donde vemos los tres botones implementados, junto al resultado de la ejecución de cada uno, el mensaje "Insertado OK" de los métodos de inserción, y la lista de clientes recuperada por el método de consulta.



Espero que estos dos últimos apartados sobre servicios web SOAP y Android os sirvan para tener un ejemplo completo, tanto de la parte servidor como de la parte cliente, que os sirva de base para crear nuevos sistemas adaptados a vuestras necesidades.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-ws-soap

Servicios Web REST: Servidor

En los dos apartados anteriores nos hemos ocupado de describir la forma de construir un sistema formado por un servicio web SOAP que accede a una base de datos externa y una aplicación Android que, a través de este servicio, es capaz de manipular dichos datos.

En este nuevo apartado vamos a crear un sistema similar, pero esta vez haciendo uso de la otra alternativa por excelencia a la hora de crear servicios web, y no es otra de utilizar servicios web tipo **REST**. Las famosas *APIs* que publican muchos de los sitios web actualmente no son más que servicios web de este tipo, aunque en la mayoría de los casos con medidas de seguridad adicionales tales como autenticación *OAuth* o similares.

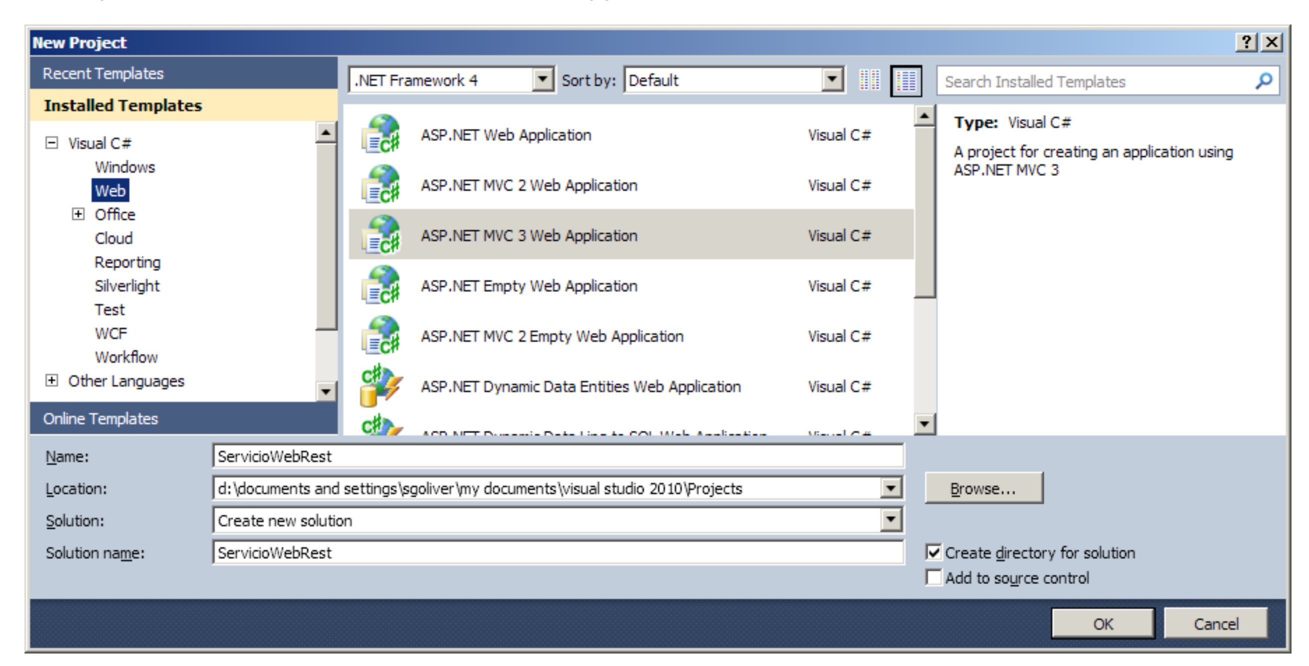
REST también se asienta sobre el protocolo HTTP como mecanismo de transporte entre cliente y servidor, ya veremos después en qué medida. Y en cuanto al formato de los datos transmitidos, a diferencia de SOAP, no se impone ninguno en concreto, aunque lo más habitual actualmente es intercambiar la información en formato XML o JSON. Ya que en el caso de SOAP utilizamos XML, en este nuevo apartado utilizaremos JSON para construir nuestro ejemplo.

También vamos a utilizar un framework distinto para construir el servicio, aunque seguiremos haciéndolo en Visual Studio y en lenguaje C#. En este caso, en vez de utilizar ASP.NET a secas, vamos a utilizar el

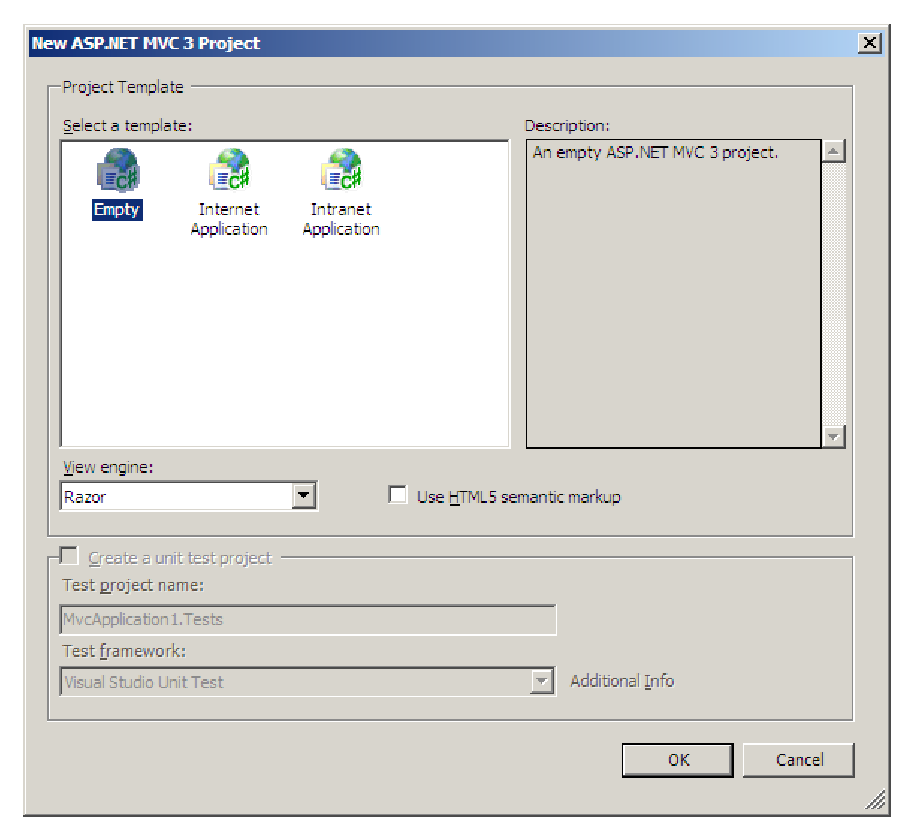
framework específico *ASP.NET MVC 3*, cuyo sistema de direccionamiento se ajusta mejor a los principios de REST, donde cada *recurso* [en nuestro caso cada cliente] debería ser accesible mediante su propia URL única. Podéis descargar MVC3 desde su [página oficial]() de Microsoft.

En este primer apartado sobre servicios REST vamos a describir la construcción del servicio web en sí, y dedicaremos un segundo apartado a explicar cómo podemos acceder a este servicio desde una aplicación Android.

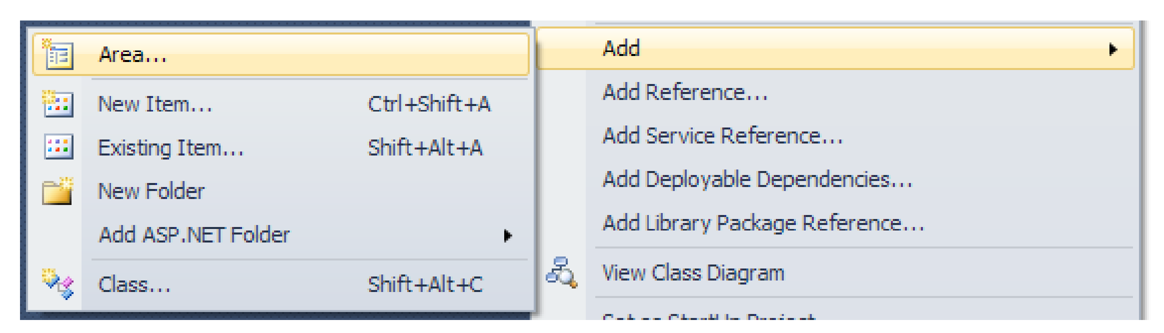
Empezamos. Lo primero que vamos a hacer será crear un nuevo proyecto en Visual Studio utilizando esta vez la plantilla llamada "ASP.NET MVC 3 Web Application", lo llamaremos "ServicioWebRest".



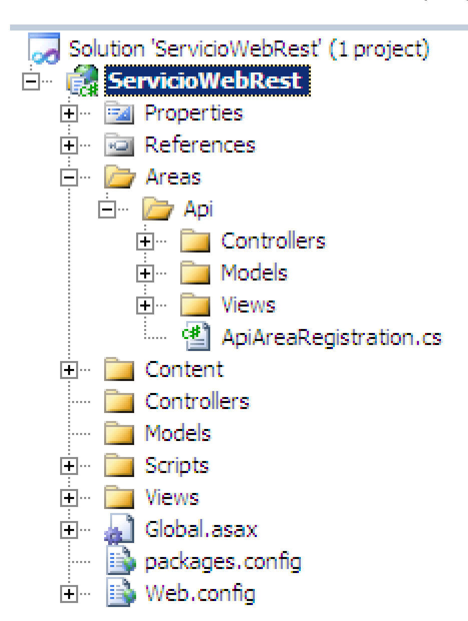
En la ventana de opciones del proyecto dejaremos todos los datos que aparecen por defecto y seleccionaremos como plantilla "Empty" para crear una aplicación vacía.



Esto debería crearnos el nuevo proyecto con la estructura de carpetas necesaria, que como veréis es bastante elaborada. En nuestro caso vamos a crear el servicio web de forma aislada del resto de la aplicación web, y para ello lo primero que vamos a hacer es añadir una nueva Area al proyecto, a la que llamaremos por ejemplo "Api", lo que nos creará una estructura de carpetas similar a la de la aplicación principal pero dentro de una carpeta independiente. Esto nos permite aislar todo el código y recursos de nuestro servicio web del resto de la aplicación web (que en nuestro caso no existirá porque no es el objetivo de este apartado, pero que podríamos crear sin problemas si lo necesitáramos).



Con esto, la estructura de nuestro proyecto será la siguiente:



Una vez que ya tenemos preparada toda la estructura de nuestro proyecto empecemos a añadir los elementos necesarios. Lo primero que vamos a crear será una nueva clase Cliente, igual que hicimos en el ejemplo anterior con SOAP. La colocaremos en la carpeta "Api/Models" y el código es el mismo que ya vimos:

```
namespace ServicioWebRest.Areas.Api.Models
{
    public class Cliente
    {
        public int Id { get; set; }
        public string Nombre { get; set; }
        public int Telefono { get; set; }
    }
}
```

El siguiente elemento a añadir será una nueva clase que contenga todas las operaciones que queramos realizar sobre nuestra base de datos de clientes. Llamaremos a la clase `ClienteManager`. En este caso sí vamos a añadir las cuatro operaciones básicas sobre clientes, y una adicional para obtener el listado completo, de forma que más tarde podamos mostrar la implementación en Android de todos los posibles

tipos de llamada al servicio. Los métodos que añadiremos serán los siguientes:

- `Cliente ObtenerCliente(int id).`
- `List<Clientes> ObtenerClientes().`
- `bool InsertarCliente(Cliente c).`
- `bool ActualizarCliente(Cliente c).`
- `bool EliminarCliente(int id).`

Los dos primeros métodos nos servirán para recuperar clientes de la base de datos, tanto por su ID para obtener un cliente concreto, como el listado completo que devolverá una lista de clientes. Los otros tres métodos permitirán insertar, actualizar y eliminar clientes a partir de su ID y los datos de entrada (si aplica). El código de todos estos métodos es análogo a los ya implementados en el caso de SOAP, por lo que no nos vamos a parar en volverlos a comentar, tan sólo decir que utilizan la API clásica de ADO.NET para el acceso a SQL Server. En cualquier caso, al final del apartado tenéis como siempre el código fuente completo para poder consultar lo que necesitéis. A modo de ejemplo veamos la implementación de los métodos `ObtenerClientes()` e `InsertarCliente()`.

```
public bool InsertarCliente(Cliente cli)
{
      SqlConnection con = new SqlConnection(cadenaConexion);

      con.Open();

      string sql = "INSERT INTO Clientes (Nombre, Telefono) VALUES (@nombre, @
telefono)";

      SqlCommand cmd = new SqlCommand(sql,con);

      cmd.Parameters.Add("@nombre", System.Data.SqlDbType.NVarChar).Value =
cli.Nombre;
      cmd.Parameters.Add("@telefono", System.Data.SqlDbType.Int).Value = cli.
Telefono;

      int res = cmd.ExecuteNonQuery();

      con.Close();

      return (res == 1);
}

public List<Cliente> ObtenerClientes()
{
      List<Cliente> lista = new List<Cliente>();

      SqlConnection con = new SqlConnection(cadenaConexion);

      con.Open();

      string sql = "SELECT IdCliente, Nombre, Telefono FROM Clientes";

      SqlCommand cmd = new SqlCommand(sql,con);

      SqlDataReader reader =
            cmd.ExecuteReader(System.Data.CommandBehavior.CloseConnection);
```

```
        while (reader.Read())
        {
            Cliente cli = new Cliente();

            cli = new Cliente();
            cli.Id = reader.GetInt32(0);
            cli.Nombre = reader.GetString(1);
            cli.Telefono = reader.GetInt32(2);

            lista.Add(cli);
        }

        reader.Close();

        return lista;
}
```

Hasta ahora, todo el código que hemos escrito es bastante genérico y nada tiene que ver con que nuestro proyecto sea de tipo MVC. Sin embargo, los dos siguientes elementos sí que están directamente relacionados con el tipo de proyecto que tenemos entre manos.

Lo siguiente que vamos a añadir será un controlador a nuestro servicio web. Este controlador (clase `ClientesController`) será el encargado de contener las diferentes acciones que se podrán llamar según la URL y datos HTTP que recibamos como petición de entrada al servicio. Para nuestro ejemplo, añadiremos tan sólo dos acciones, una primera dirigida a gestionar todas las peticiones que afecten a un único cliente (insertar, actualizar, eliminar y obtener por ID), y otra que trate la petición del listado completo de clientes. Las llamaremos `Clientes()` y `Cliente()` respectivamente. Estas acciones harán uso de una instancia de la clase `ClienteManager` creada anteriormente para realizar las acciones necesarias contra la base de datos. Cada acción será también responsable de formatear sus resultados al formato de comunicación que hayamos elegido, en nuestro caso JSON.

La acción Clientes es muy sencilla, se limitará a llamar al método `ObtenerClientes()` y formatear los resultados como JSON. Para hacer esto último basta con crear directamente un objeto `JsonResult` llamado al método `Json()` pasándole como parámetro de entrada el objeto a formatear. Todo esto se reduce a una sola linea de código:

```
[HttpGet]
public JsonResult Clientes()
{
    return Json(this.clientesManager.ObtenerClientes(),
                JsonRequestBehavior.AllowGet);
}
```

Habréis notado también que hemos precedido el método con el atributo `[HttpGet]`. Para intentar explicar esto me hace falta seguir hablando de los principios de diseño de REST. Este tipo de servicios utiliza los propios tipos de petición definidos por el protocolo HTTP para diferenciar entre las operaciones a realizar por el servicio web. Así, el propio tipo de petición HTTP realizada (`GET`, `POST`, `PUT` o `DELETE`), junto con la dirección URL especificada en la llamada, nos determinará la operación a ejecutar por el servicio web. En el caso ya visto, el atributo `[HttpGet]` nos indica que dicho método se podrá ejecutar al recibirse una petición de tipo GET.

Entenderemos todo esto mejor ahora cuando veamos el código de la acción `Cliente()`. En esta acción, dependiente del tipo de petición HTTP recibida, tendremos que llamar a un método u otro del servicio web. Así, usaremos `POST` para las inserciones de clientes, `PUT` para las actualizaciones, `GET` para la consulta por ID y `DELETE` para las eliminaciones. En este caso no precedemos el método por ningún atributo, ya que la misma acción se encargará de tratar diferentes tipos de petición.

```
public JsonResult Cliente(int? id, Cliente item)
{
    switch (Request.HttpMethod)
    {
        case "POST":
            return Json(clientesManager.InsertarCliente(item));
        case "PUT":
            return Json(clientesManager.ActualizarCliente(item));
        case "GET":
            return Json(clientesManager.ObtenerCliente(id.
GetValueOrDefault()),
                        JsonRequestBehavior.AllowGet);
        case "DELETE":
            return Json(clientesManager.EliminarCliente(id.
GetValueOrDefault()));
    }

    return Json(new { Error = true, Message = "Operación HTTP desconocida" });
}
```

Algunos de vosotros seguro que os estáis preguntando cómo distinguirá el servicio cuándo llamar a la acción `Clientes()` para obtener el listado completo, o a la acción `Cliente()` para obtener un único cliente por su ID, ya que para ambas operaciones hemos indicado que se recibirá el tipo de petición http `GET`.

Pues bien, aquí es donde nos va a ayudar el último elemento a añadir al servicio web. Realmente no lo añadiremos, sino que lo modificaremos, ya que es un fichero que ya ha creado Visual Studio por nosotros. Se trata de la clase `ApiAreaRegistration`. La función de esta clase será la de dirigir las peticiones recibidas hacia una acción u otra del controlador según la URL utilizada al realizarse la llamada al servicio web.

En nuestro caso de ejemplo, vamos a reconocer dos tipos de URL. Una de ellas para acceder a la lista completa de cliente, y otra para realizar cualquier acción sobre un cliente en concreto:

- Lista de clientes: *http://servidor/Api/Clientes*
- Operación sobre cliente: *http://servidor/Api/Clientes/Cliente/id_del_cliente*

Cada uno de estos patrones tendremos que registrarlos mediante el método `MapRoute()` dentro del método `RegisterArea()` que ya tendremos creado dentro de la clase `ApiAreaRegistration`. Así, para registrar el primer tipo de URL haremos lo siguiente:

```
context.MapRoute(
    "AccesoClientes",
    "Api/Clientes",
    new
    {
        controller = "Clientes",
        action = "Clientes"
    }
);
```

Como primer parámetro de `MapRoute()` indicamos un nombre descriptivo para el patrón de URL. El segundo parámetro es el patrón en sí, que en este caso no tiene partes variables. Por último indicamos el controlador al que se dirigirán las peticiones que sigan este patrón eliminando el sufijo "`Controller`" (en nuestro caso será el controlador `ClientesController`) y la acción concreta a ejecutar dentro de dicho controlador (en nuestro caso la acción `Clientes()`).

Para el segundo tipo de URL será muy similar, con la única diferencia de que ahora habrá una parte final

variable que se corresponderá con el ID del cliente y que asignaremos al parámetro "`id`" de la acción. En este caso además, dirigiremos la petición hacia la acción `Cliente()`, en vez de `Clientes()`.

```
context.MapRoute(
    "AccesoCliente",
    "Api/Clientes/Cliente/{id}",
    new
    {
        controller = "Clientes",
        action = "Cliente",
        id = UrlParameter.Optional
    }
);
```

Como todo esto en cuenta, y por recapitular un poco, las posibles llamadas a nuestro servicio serán las siguientes:

`GET  /Api/Clientes`

Recuperará el listado completo de clientes y lo devolverá en formato JSON.

`GET  /Api/Clientes/Cliente/3`

Recuperará el cliente con el ID indicado en la URL y lo devolverá en formato JSON.

`POST  /Api/Clientes/Cliente  { Nombre:"nombre", Telefono:1234 }`

Insertará un nuevo cliente con los datos aportados en la petición en formato JSON.

`PUT  /Api/Clientes/Cliente/3  { Id:3, Nombre:"nombre", Telefono:1234 }`

Actualizará el cliente con el ID indicado en la URL con los datos aportados en la petición en formato JSON.

`DELETE  /Api/Clientes/Cliente/3`

Eliminará el cliente con el ID indicado en la URL.

Llegados aquí, tan sólo tenemos que ejecutar nuestro proyecto y esperar a que se abra el navegador web. En principio no se mostrará un error por no encontrar la página principal de la aplicación, ya que no hemos creado ninguna, pero nos asegurará que el servidor de prueba está funcionando, por lo que nuestro servicio debería responder a peticiones.

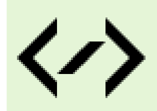
Así, si escribimos en la barra de direcciones por ejemplo la siguiente dirección (el puerto puede variar):

`http://localhost:1234/Api/Clientes/Cliente/4`

deberíamos recibir un fichero en formato JSON que contuviera los datos del cliente con ID = 4 de nuestra base de datos. Sería un fichero con contenido similar al siguiente:

`{"Id":4,"Nombre":"cliente4","Telefono":4444}`

En el siguiente apartado veremos cómo construir una aplicación Android capaz de acceder a este servicio y procesar los resultados recibidos.



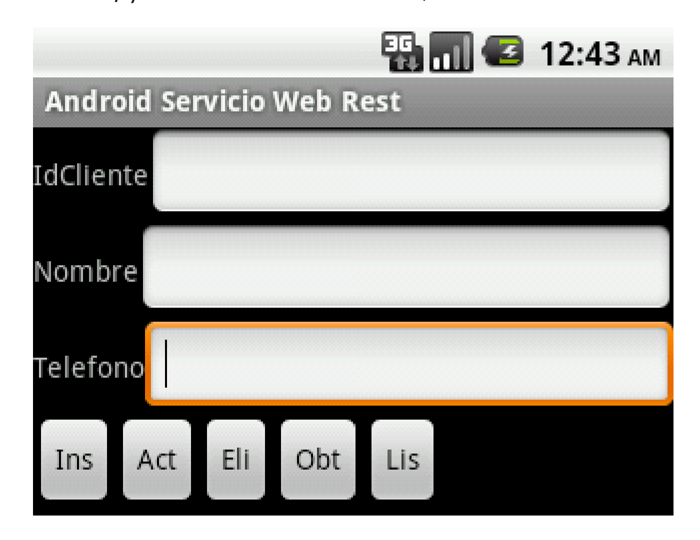
Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/dotnet-serviciowebrest

Servicios Web REST: Cliente

En el apartado anterior dedicado a los servicios web REST hemos visto cómo crear fácilmente un servicio de este tipo utilizando el framework ASP.NET MVC 3. En esta segunda parte vamos a describir cómo podemos construir una aplicación Android que acceda a este servicio web REST.

Y tal como hicimos en el caso de SOAP, vamos a crear una aplicación de ejemplo que llame a las distintas funciones de nuestro servicio web. En este caso la aplicación se compondrá de 5 botones, uno por cada una de las acciones que hemos implementado en el servicio web (insertar, actualizar, eliminar, recuperar un cliente, y listar todos los clientes).



A diferencia del caso de SOAP, en esta ocasión no vamos a utilizar ninguna librería externa para acceder al servicio web, ya que Android incluye todo lo necesario para realizar la conexión y llamada a los métodos del servicio, y tratamiento de resultados en formato JSON.

Como ya hemos comentado, al trabajar con servicios web de tipo REST, las llamadas al servicio no se harán a través de una única URL, sino que se determinará la acción a realizar según la URL accedida y la acción HTTP utilizada para realizar la petición (`GET`, `POST`, `PUT` o `DELETE`). En los siguientes apartados veremos uno a uno la implementación de estos botones.

Insertar un nuevo cliente

Como ya comentamos en el apartado anterior, la inserción de un nuevo cliente la realizaremos a través de la siguiente URL:

```
http://10.0.2.2:2731/Api/Clientes/Cliente
```

Utilizaremos la acción http POST y tendremos que incluir en la petición un objeto en formato JSON que contenga los datos del nuevo cliente (tan sólo Nombre y Teléfono, ya que el ID se calculará automáticamente). El formato de este objeto de entrada será análogo al siguiente:

```
{Nombre:"cccc", Telefono:12345678}
```

Pues bien, para conseguir esto comenzaremos por crear un nuevo objeto `HttpClient`, que será el encargado de realizar la comunicación HTTP con el servidor a partir de los datos que nosotros le proporcionemos. Tras esto crearemos la petición `POST` creando un nuevo objeto `HttpPost` e indicando la URL de llamada al

servicio. Modificaremos mediante `setHeader()` el atributo http `content-type` para indicar que el formato de los datos que utilizaremos en la comunicación, que como ya indicamos será JSON (cuyo *MIME-Type* correspondiente es "*application/json*").

```
HttpClient httpClient = new DefaultHttpClient();

HttpPost post =
    new HttpPost("http://10.0.2.2:2731/Api/Clientes/Cliente");

post.setHeader("content-type", "application/json");
```

El siguiente paso será crear el objeto JSON a incluir con la petición, que deberá contener los datos del nuevo cliente a insertar. Para ello creamos un nuevo objeto `JSONObject` y le añadimos mediante el método `put()` los dos atributos necesarios (nombre y teléfono) con sus valores correspondientes, que los obtenemos de los cuadros de texto de la interfaz, llamados `txtNombre` y `txtTelefono`.

Por último asociaremos este objeto JSON a nuestra petición HTTP convirtiéndolo primero al tipo `StringEntity` e incluyéndolo finalmente en la petición mediante el método `setEntity()`.

```
//Construimos el objeto cliente en formato JSON
JSONObject dato = new JSONObject();

dato.put("Nombre", txtNombre.getText().toString());
dato.put("Telefono", Integer.parseInt(txtTelefono.getText().toString()));

StringEntity entity = new StringEntity(dato.toString());
post.setEntity(entity);
```

Una vez creada nuestra petición HTTP y asociado el dato de entrada, tan sólo nos queda realizar la llamada al servicio mediante el método `execute()` del objeto `HttpClient` y recuperar el resultado mediante `getEntity()`. Este resultado lo recibimos en forma de objeto `HttpEntity`, pero lo podemos convertir fácilmente en una cadena de texto mediante el método estático `EntityUtils.toString()`.

```
HttpResponse resp = httpClient.execute(post);
String respStr = EntityUtils.toString(resp.getEntity());

if(respStr.equals("true"))
    lblResultado.setText("Insertado OK.");
```

En nuestro caso, el método de inserción devuelve únicamente un valor booleano indicando si el registro se ha insertado correctamente en la base de datos, por lo que tan sólo tendremos que verificar el valor de este *booleano* ("true" o "false") para conocer el resultado de la operación, que mostraremos en la interfaz en una etiqueta de texto llamada `lblResultado`.

Como ya dijimos en el apartado anterior sobre servicios SOAP, a partir de la versión 3 de Android no se permite realizar operaciones de larga duración dentro del hilo principal de la aplicación, entre ellas conexiones a internet como estamos haciendo en esta ocasión. Para solucionar este problema y que la aplicación funcione correctamente en todas las versiones de Android debemos hacer la llamada al servicio mediante una tarea asíncrona, o `AsynTask`, que se ejecute en segundo plano. Una vez más no entraré en detalles porque el curso contiene un capítulo dedicado exclusivamente a este tema. A modo de ejemplo, el código anterior trasladado a una `AsyncTask` quedaría de la siguiente forma:

```
private class TareaWSInsertar extends AsyncTask<String,Integer,Boolean> {

    protected Boolean doInBackground(String... params) {

      boolean resul = true;

      HttpClient httpClient = new DefaultHttpClient();

      HttpPost post = new
             HttpPost("http://10.0.2.2:2731/Api/Clientes/Cliente");

      post.setHeader("content-type", "application/json");

      try {
            //Construimos el objeto cliente en formato JSON
            JSONObject dato = new JSONObject();

            dato.put("Nombre", params[0]);
            dato.put("Telefono", Integer.parseInt(params[1]));

            StringEntity entity = new StringEntity(dato.toString());
            post.setEntity(entity);

            HttpResponse resp = httpClient.execute(post);
            String respStr = EntityUtils.toString(resp.getEntity());

            if(!respStr.equals("true"))
                  resul = false;
        }
        catch(Exception ex) {
            Log.e("ServicioRest","Error!", ex);
            resul = false;
        }

        return resul;
    }

    protected void onPostExecute(Boolean result) {

      if (result) {
            lblResultado.setText("Insertado OK.");
      }
    }
}
```

Y la llamada a la tarea asíncrona desde el evento `onClick` del botón de *Insertar* sería tan sencilla como ésta:

```
btnInsertar.setOnClickListener(new OnClickListener() {

      @Override
      public void onClick(View v) {
            TareaWSInsertar tarea = new TareaWSInsertar();
            tarea.execute(
                  txtNombre.getText().toString(),
                  txtTelefono.getText().toString());
      }
});
```

Actualizar un cliente existente

La URL utilizada para la actualización de clientes será la misma que la anterior:

`http://10.0.2.2:2731/Api/Clientes/Cliente`

Pero en este caso, el objeto JSON a enviar como entrada deberá contener no sólo los nuevos valores de nombre y teléfono sino también el ID del cliente a actualizar, por lo que tendría una estructura análoga a la siguiente:

`{Id:123, Nombre:"cccc", Telefono:12345678}`

Para actualizar el cliente procederemos de una forma muy similar a la ya comentada para la inserción, con las únicas diferencias de que en este caso la acción HTTP utilizada será `PUT` (objeto `HttpPut`) y que el objeto JSON de entrada tendrá el campo ID adicional.

```
HttpClient httpClient = new DefaultHttpClient();

HttpPut put = new HttpPut("http://10.0.2.2:2731/Api/Clientes/Cliente");
put.setHeader("content-type", "application/json");

try
{
    //Construimos el objeto cliente en formato JSON
    JSONObject dato = new JSONObject();

    dato.put("Id", Integer.parseInt(txtId.getText().toString()));
    dato.put("Nombre", txtNombre.getText().toString());
    dato.put("Telefono", Integer.parseInt(txtTelefono.getText().toString()));

    StringEntity entity = new StringEntity(dato.toString());
    put.setEntity(entity);

        HttpResponse resp = httpClient.execute(put);
        String respStr = EntityUtils.toString(resp.getEntity());

        if(respStr.equals("true"))
            lblResultado.setText("Actualizado OK.");
}
catch(Exception ex)
{
        Log.e("ServicioRest","Error!", ex);
}
```

Eliminación de un cliente

La eliminación de un cliente la realizaremos a través de la URL siguiente:

`http://10.0.2.2:2731/Api/Clientes/Cliente/id_cliente`

donde id_cliente será el ID del cliente a eliminar. Además, utilizaremos la acción http `DELETE` (objeto `HttpDelete`) para identificar la operación que queremos realizar. En este caso no será necesario pasar ningún objeto de entrada junto con la petición, por lo que el código quedará aún más sencillo que los dos casos anteriores.

```
HttpClient httpClient = new DefaultHttpClient();

String id = txtId.getText().toString();

HttpDelete del =
    new HttpDelete("http://10.0.2.2:2731/Api/Clientes/Cliente/" + id);

del.setHeader("content-type", "application/json");

try {
        HttpResponse resp = httpClient.execute(del);
        String respStr = EntityUtils.toString(resp.getEntity());

        if(respStr.equals("true"))
            lblResultado.setText("Eliminado OK.");
}
catch(Exception ex) {
        Log.e("ServicioRest","Error!", ex);
}
```

Como podéis ver, al principio del método obtenemos el ID del cliente desde la interfaz de la aplicación y lo concatenamos con la URL base para formar la URL completa de llamada al servicio.

Obtener un cliente

Esta operación es un poco distinta a las anteriores, ya que en este caso el resultado devuelto por el servicio será un objeto JSON y no un valor simple como en los casos anteriores. Al igual que en el caso de eliminación de clientes, la URL a utilizar será del tipo:

`http://10.0.2.2:2731/Api/Clientes/Cliente/id_cliente`

En este caso utilizaremos un tipo de petición http `GET` (objeto `HttpGet`) y la forma de realizar la llamada será análoga a las anteriores. Donde aparecerán las diferencias será a la hora de tratar el resultado devuelto por el servicio tras llamar al método `getEntity()`. Lo que haremos será crear un nuevo objeto `JSONObject` a partir del resultado textual de `getEntity()`. Hecho esto, podremos acceder a los atributos del objeto utilizando para ello los métodos `get()` correspondientes, según el tipo de cada atributo (`getInt()`, `getString()`, etc). Tras esto mostraremos los datos del cliente recuperado en la etiqueta de resultados de la interfaz (`lblResultados`).

```
HttpClient httpClient = new DefaultHttpClient();

String id = txtId.getText().toString();

HttpGet del =
    new HttpGet("http://10.0.2.2:2731/Api/Clientes/Cliente/" + id);

del.setHeader("content-type", "application/json");

try {
        HttpResponse resp = httpClient.execute(del);
        String respStr = EntityUtils.toString(resp.getEntity());

        JSONObject respJSON = new JSONObject(respStr);

        int idCli = respJSON.getInt("Id");
        String nombCli = respJSON.getString("Nombre");
        int telefCli = respJSON.getInt("Telefono");

        lblResultado.setText("" + idCli + "-" + nombCli + "-" + telefCli);
}
catch(Exception ex) {
        Log.e("ServicioRest","Error!", ex);
}
```

Una vez más como podéis comprobar el código es muy similar al ya visto para el resto de operaciones.

Obtener listado completo de clientes

Por último vamos a ver cómo podemos obtener el listado completo de clientes. El interés de esta operación está en que el resultado recuperado de la llamada al servicio será un array de objetos de tipo cliente, por supuesto en formato JSON. La acción http utilizada será una vez más la acción `GET`, y la URL para recuperar el listado de clientes será:

`http://10.0.2.2:2731/Api/Clientes`

De nuevo, la forma de llamar al servicio será análoga a las anteriores hasta la llamada a `getEntity()` para recuperar los resultados. En esta ocasión, dado que recibimos un array de elementos, convertiremos este resultado a un objeto `JSONArray`, y hecho esto podremos acceder a cada uno de los elementos del array mediante una llamada a `getJSONObject()`, al que iremos pasando el índice de cada elemento. Para saber cuántos elementos contiene el array podremos utilizar el método `length()` del objeto `JSONArray`. Por último, el acceso a los atributos de cada elemento del array lo realizamos exactamente igual como ya lo hicimos en la operación anterior de obtención de cliente por ID.

```
HttpClient httpClient = new DefaultHttpClient();

HttpGet del =
    new HttpGet("http://10.0.2.2:2731/Api/Clientes");

del.setHeader("content-type", "application/json");

try
{
        HttpResponse resp = httpClient.execute(del);
        String respStr = EntityUtils.toString(resp.getEntity());

        JSONArray respJSON = new JSONArray(respStr);

        String[] clientes = new String[respJSON.length()];

        for(int i=0; i<respJSON.length(); i++)
        {
            JSONObject obj = respJSON.getJSONObject(i);

            int idCli = obj.getInt("Id");
            String nombCli = obj.getString("Nombre");
            int telefCli = obj.getInt("Telefono");

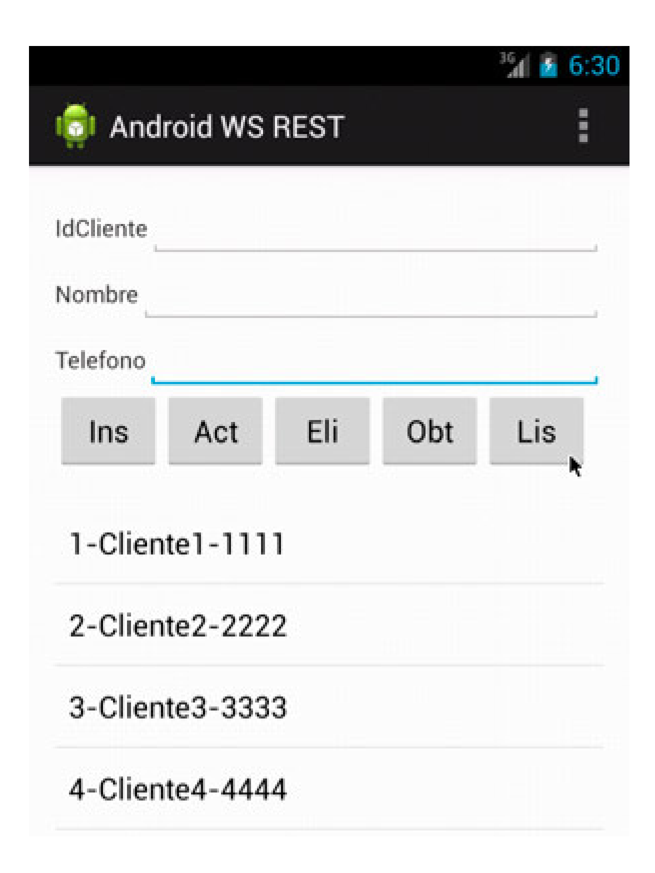
            clientes[i] = "" + idCli + "-" + nombCli + "-" + telefCli;
        }

        //Rellenamos la lista con los resultados
        ArrayAdapter<String> adaptador =
                new ArrayAdapter<String>(ServicioWebRest.this,
                android.R.layout.simple_list_item_1, clientes);

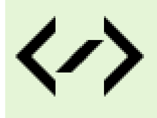
    lstClientes.setAdapter(adaptador);
}
catch(Exception ex)
{
        Log.e("ServicioRest","Error!", ex);
}
```

Tras obtener nuestro array de clientes, para mostrar los resultados hemos añadido a la interfas de nuestra aplicación de ejemplo un control tipo `ListView` (llamado `lstClientes`) que hemos rellenado a través de su adaptador con los datos de los clientes recuperados.

A modo de ejemplo, en la siguiente imagen puede verse el resultado de ejecutar la operación de listado completo de clientes:



Y con esto hemos terminado. Espero haber ilustrado con claridad en los dos últimos apartados la forma de construir servicios web tipo REST mediante ASP.NET y aplicaciones cliente Android capaces de acceder a dichos servicios.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-ws-rest

15

Notificaciones Push

XV. Notificaciones Push

Introducción a Google Cloud Messaging

Aprovechando que el servicio de mensajería push de Google ha salido de su fase beta recientemente, y que ha sufrido algunos cambios con respecto a su anterior versión (C2DM), voy a dedicar los próximos apartados a describir qué es y cómo utilizar este servicio. En este primer apartado haremos una introducción al servicio y comentaremos la forma de registrarnos para poder utilizarlo (no preocuparos, es gratuito), y en los dos siguientes veremos cómo implementar las aplicaciones servidor (una vez más en ASP.NET) y cliente (en Android). Empecemos.

En algunas ocasiones, nuestras aplicaciones móviles necesitan contar con la capacidad de notificar al usuario determinados eventos que ocurren fuera del dispositivo, normalmente en una aplicación web o servicio online alojado en un servidor externo. Como ejemplo de esto podríamos pensar en las notificaciones que nos muestra nuestro móvil cuando se recibe un nuevo correo electrónico en nuestro servidor de correo habitual.

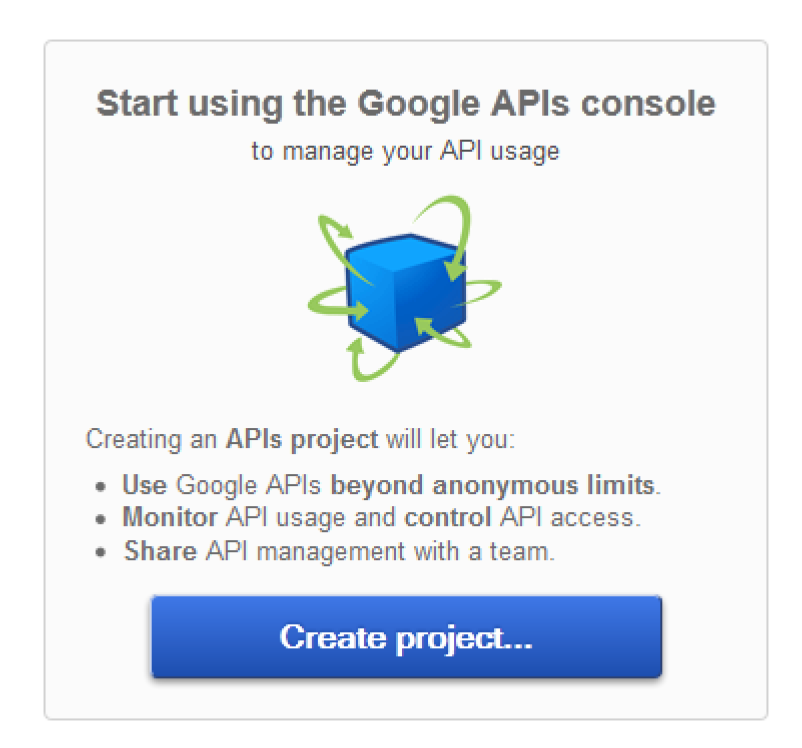
Para conseguir esto se nos podrían ocurrir varias soluciones, por ejemplo mantener abierta una conexión permanente con el servidor de forma que éste le pudiera comunicar inmediatamente cualquier nuevo evento a nuestra aplicación. Esta técnica, aunque es viable y efectiva, requiere de muchos recursos abiertos constantemente en nuestro dispositivo, aumentando por tanto el consumo de CPU, memoria y datos de red necesarios para ejecutar la aplicación. Otra solución utilizada habitualmente sería hacer que nuestra aplicación móvil revise de forma periódica en el servidor si existe alguna novedad que notificar al usuario. Esto se denomina polling, y requiere muchos menos recursos que la opción anterior, pero tiene un inconveniente que puede ser importante según el objetivo de nuestra aplicación: cualquier evento que se produzca en el servidor no se notificará al usuario hasta la próxima consulta al servidor que haga la aplicación cliente, que podría ser mucho tiempo después.

Para solventar este problema Google introdujo en Android, a partir de la versión 2.2 (Froyo), la posibilidad de implementar notificaciones de tipo push, lo que significa que es el servidor el que inicia el proceso de notificación, pudiendo realizarla en el mismo momento que se produce el evento, y el cliente se limita a "esperar" los mensaje sin tener que estar periodicamente consultando al servidor para ver si existen novedades, y sin tener que mantener una conexión permanentemente abierta con éste. En la arquitectura de Google, todo esto se consigue introduciendo un nuevo actor en el proceso, un **servidor de mensajería push o *cloud to device*** (que se traduciría en algo así como "mensajes de la nube al dispositivo"), que se situaría entre la aplicación web y la aplicación móvil. Este servidor intermedio se encargará de recibir las notificaciones enviadas desde las aplicaciones web y hacerlas llegar a las aplicaciones móviles instaladas en los dispositivos correspondientes. Para ello, deberá conocer la existencia de ambas aplicaciones, lo que se consigue mediante un "protocolo" bien definido de registros y autorizaciones entre los distintos actores que participan en el proceso. Veremos más adelante cómo implementar este proceso.

Este servicio de Google recibió en sus comienzos las siglas **C2DM** (Cloud to Device Messaging), pero recientemente y coincidiendo con su salida de fase beta ha modificado su nombre a **GCM** (Google Cloud Messaging). Y lo primero que debemos comentar es la forma de darnos de alta en el servicio, que a pesar de ser gratuito requiere de un proceso previo de registro y la generación de una ApiKey, algo similar a lo que ya vimos al hablar de la utilización de mapas en Android. Para hacer esto debemos acceder a la consola de APIs de Google, en siguiente dirección:

`https://code.google.com/apis/console`

Suponiendo que es la primera vez que accedemos a esta consola, nos aparecerá la pantalla de bienvenida y la opción de crear un nuevo proyecto.

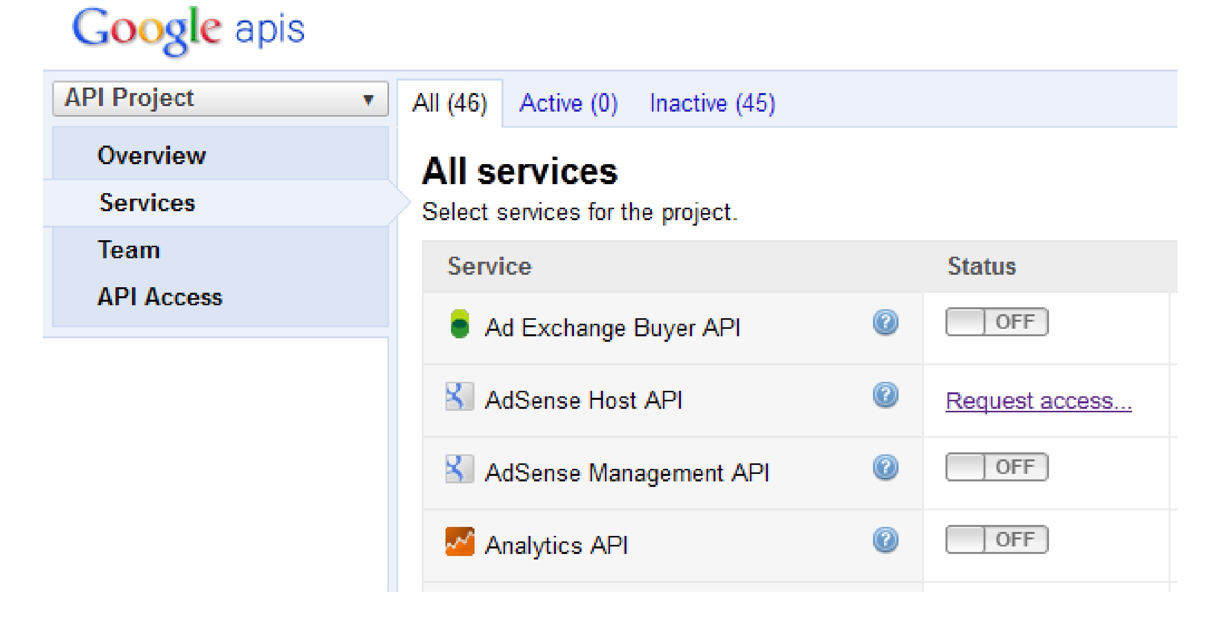


Una vez creado el proyecto el navegador te redirige a una dirección similar a ésta:



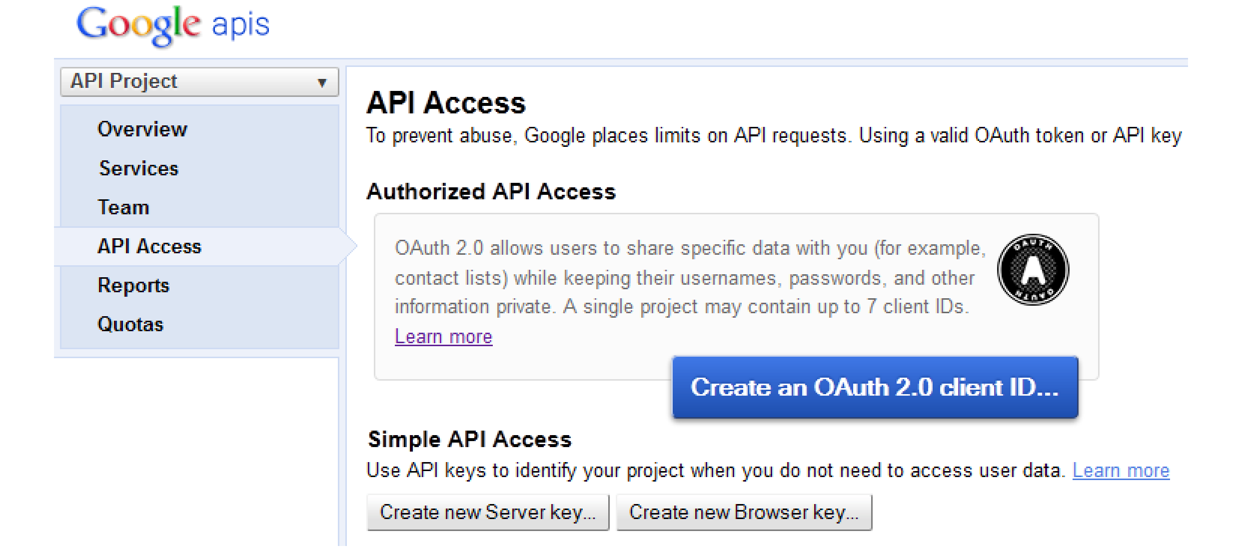
Al número que aparece tras la etiqueta "`#project:`" lo llamaremos "Sender ID" y debemos anotarlo ya que nos hará falta más adelante como identificador único de la aplicación web emisora de nuestros mensajes.

Una vez creado el nuevo proyecto vamos a activar el servicio GCM accediendo al menú "Services" y pulsando el botón ON/OFF asociado al servicio llamado "*Google Cloud Messaging*".

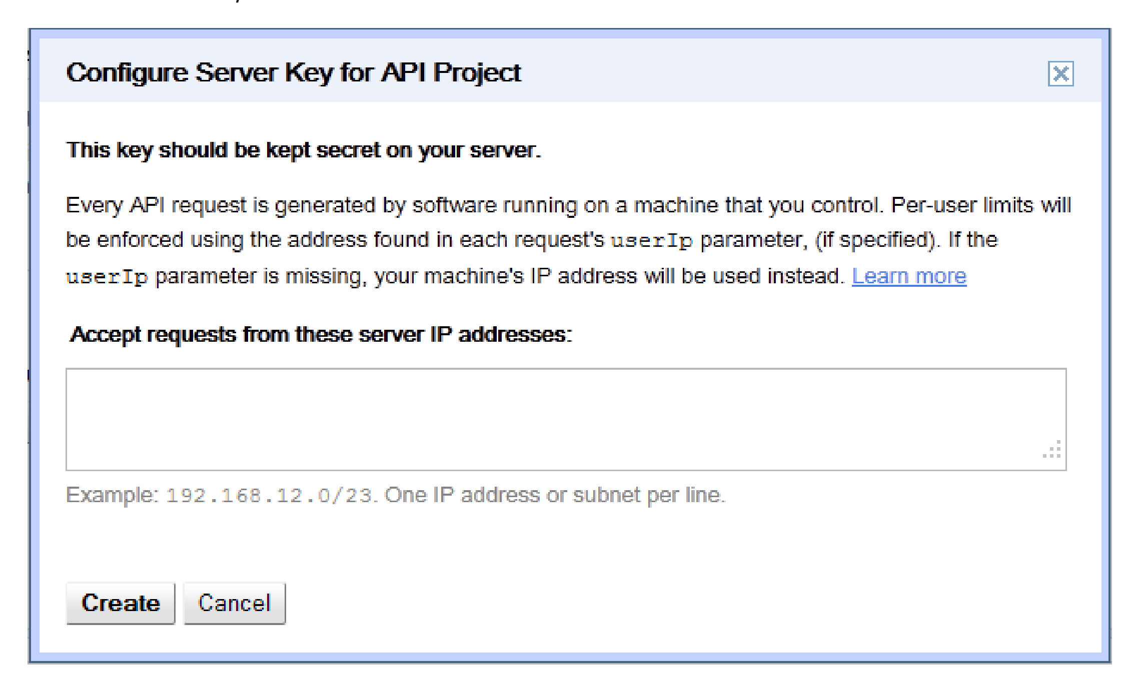


Se nos presentará entonces una ventana donde tendremos que aceptar las condiciones del servicio y tras ésto el servicio quedará activado, aunque aún nos faltará un último paso para poder utilizarlo. Como en el caso de la utilización de la API de Google Maps, para hacer uso del servicio GCM tendremos que generar una *ApiKey* que nos sirva posteriormente como identificación de acceso. Para ello accedemos al menú "Api

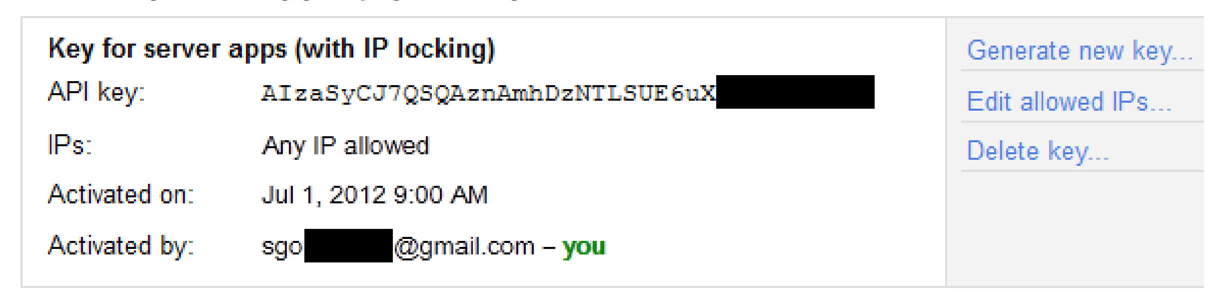
Access" y pulsaremos sobre el botón "Create new Server Key...".



Nos aparecerá un diálogo llamado "Configure Server Key for API Project", que aceptaremos sin más pulsando el botón "Create", sin necesidad de rellenar nada.

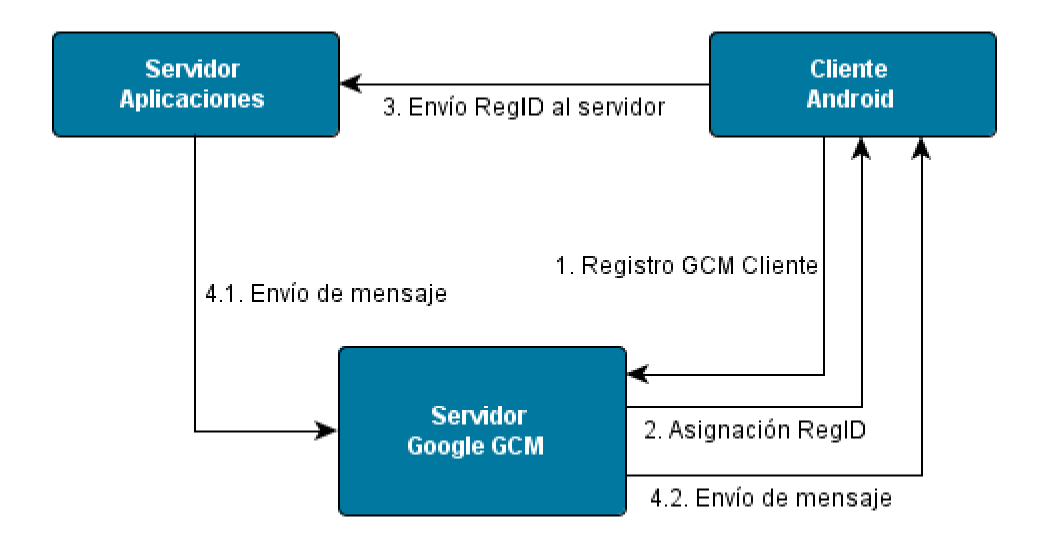


Y ya tendríamos creada nuestra API Key, que aparecerá en la sección "Simple API Access" con el título "Key for server apps (with IP locking)".



Con esto ya nos habríamos registrado correctamente en el servicio GCM y habríamos generado nuestra API Key para identificarnos, con lo que estaríamos en disposición de construir nuestras aplicaciones cliente y servidor, algo que veremos en los dos próximos apartados. En éste nos pararemos un poco más para hacer una descripción a grandes rasgos de la arquitectura que tendrá nuestro sistema y de cómo fluirá la información entre cada uno de los servicios y aplicaciones implicados.

Como hemos indicado antes, para asegurar la correcta comunicación entre los tres sistemas se hace necesario un protocolo de registros y autorizaciones que proporcione seguridad y calidad a todo el proceso. Este proceso se resume en el siguiente gráfico, que intentaré explicar a continuación.



Comentemos brevemente cada uno de los pasos indicados en el diagrama.

El primer paso, aunque no aparece en el gráfico, sería la autenticación de nuestra aplicación web en el servicio GCM. En la anterior versión del servicio GCM (llamada C2DM) esto debía hacerse mediante la utilización de otra API de Google llamada *ClientLogin*, o a través del protocolo OAuth 2.0, ambas dirigidas a obtener un token de autorización que debía enviarse posteriormente en el resto de llamadas al servicio. Sin embargo, con la llegada de Google Cloud Messaging, esto se ha simplificado mediante la obtención y uso de una *API Key*, que es precisamente lo que hemos comentado unos párrafos más arriba. Como pasaba con el token de autorización, nuestra nueva API Key deberá acompañar a cada una de las llamadas que hagamos al servicio GCM desde nuestra aplicación web.

Los siguientes pasado, ya sí mostrados en el diagrama, serían los siguientes:

1. El siguiente paso es el equivalente al ya comentado para el servidor pero esta vez desde el punto de vista de la aplicación cliente. La aplicación Android debe registrarse en los servidores GCM como cliente capaz de recibir mensajes desde dicho servicio. Para esto es necesario que el dispositivo/emulador cumplan una serie de requisitos:

    a. Disponer de Android 2.2 o superior.

    b. Tener configurada una cuenta de Google en el dispositivo o emulador. Configurable desde "Ajustes / Cuentas y sincronización".

    c. Si se trata de un dispositivo real debe estar instalada la Google Play Store. Por el contrario si estamos ejecutando la aplicación desde el emulador bastará con usar un *target* que incluya las APIs de Google.

2. Si el registro se finaliza correctamente se recibirá un código de registro (Registration ID) que la aplicación cliente deberá conservar. Además, la aplicación Android deberá estar preparada para recibir periódicamente refrescos de este código de registro, ya que es posible que el servidor GCM invalide periódicamente este ID, genere uno nuevo y lo vuelva a notificar a la aplicación cliente.

3. Este nuevo paso consiste en enviar, desde la aplicación cliente a la aplicación servidor, el código de registro GCM recibido, el cual hará las veces de identificador único del cliente en el servidor de forma que éste pueda indicar más tarde el dispositivo móvil concreto al que desea enviar un mensaje. La aplicación servidora tendrá que ser capaz por tanto de almacenar y mantener todos los ID de registro de los distintos dispositivos móviles que se registren como clientes capaces de recibir mensajes.

4. El último paso será obviamente el envío en sí de un mensaje desde el servidor hasta un cliente determinado, algo que se hará a través del servidor GCM (paso 4.1) y desde éste se dirigirá al cliente concreto que debe recibirlo (paso 4.2).

Como se puede comprobar, el procedimiento es relativamente complejo, aunque bastante intuitivo. En los próximos apartados veremos cómo implementar cada uno de ellos. Una vez más, para nuestro ejemplo utilizaremos una aplicación ASP.NET como aplicación servidora, con SQL Server a modo de almacén de datos, y aprovechando que ya hemos visto como crear servicios web SOAP y llamarlos desde aplicaciones Android, vamos a utilizar uno de éstos para la comunicación entre cliente y servidor (paso 3).

Implementación del Servidor

En el apartado anterior hicimos una introducción al servicio Google Cloud Messaging (GCM), vimos cómo registrarnos y obtener la API Key necesaria para enviar mensajes y describimos a alto nivel la arquitectura que tendrá un sistema capaz de gestionar mensajería de tipo *push* a través de este servicio de Google. Este segundo apartado lo vamos a dedicar a la implementación de una aplicación web capaz de enviar mensajes o notificaciones push a dispositivos Android. En el próximo apartado veremos cómo desarrollar la aplicación Android cliente capaz de recibir estos mensajes.

Como ya viene siendo habitual en el curso, el sistema elegido para desarrollar la aplicación web será ASP.NET, utilizando C# como lenguaje, y SQL Server como base de datos.

Como ya comentamos en el apartado anterior, la aplicación web será responsable de las siguientes tareas:

1. Almacenar y mantener el listado de dispositivos cliente que podrán recibir mensajes.

2. Enviar los mensajes a los clientes a través del servicio GCM de Google.

En cuanto al punto 1, la aplicación deberá ser capaz de recibir los datos de registro de cada cliente que se "dé de alta" para recibir mensajes y almacenarlos en la base de datos. Esto lo haremos mediante la creación de un servicio web que exponga un método capaz de recoger y almacenar los datos de registro de un cliente. La aplicación Android se conectará directamente a este servicio web y llamará al método con sus datos identificativos para registrarse como cliente capaz de recibir notificaciones. Por supuesto que para esto se podría utilizar cualquier otro mecanismo distinto a servicios web, por ejemplo una simple petición HTTP al servidor pasando los datos como parámetros, pero no nos vendrá mal para seguir practicando con servicios web en android, que en este caso será de tipo SOAP.

Por su lado, el punto 2 lo resolveremos a modo de ejemplo con una página web sencilla en la que podamos indicar el nombre de usuario de cualquiera de los dispositivos registrados en la base de datos y enviar un mensaje de prueba a dicho cliente.

Vamos a empezar creando la base de datos, aunque no nos detendremos mucho porque ya vimos el procedimiento por ejemplo en el primer apartado dedicado a servicios web SOAP. Tan sólo decir que crearemos una nueva base de datos llamada `DBUSUARIOS`, que tendrá una sola tabla `Usuarios` dos campos: `NombreUsuario` y `CodigoC2DM`, el primero de ellos destinado a almacenar un nombre de usuario identificativo de cada cliente registrado, y el segundo para almacenar el `RegistrationID` de GCM recibido desde dicho cliente a través del servicio web (recomiendo consultar el apartado anterior para entender bien todo este "protocolo" requerido por GCM).

Una vez creada la base de datos vamos a crear en Visual Studio 2010 un nuevo proyecto C# de tipo "ASP.NET Web Application" al que llamaremos "GCMServer", y añadiremos a este proyecto un nuevo componente de tipo "Web Service" llamado "ServicioRegistroGCM.asmx". Todo este procedimiento también se puede

consultar en el apartado sobre servicios web SOAP en Android.

Añadiremos un sólo método web al servicio, al que llamaremos RegistroCliente() y que recibirá como hemos comentado 2 parámetros: el nombre de usuario y el ID de registro del cliente en GCM. El método se limitará a realizar el INSERT o UPDATE correspondiente con estos dos datos en la base de datos que hemos creado de usuarios.

```
 [WebMethod]
public int RegistroCliente(string usuario, string regGCM)
{
     SqlConnection con =
         new SqlConnection(
               @"Data Source=SGOLIVERPC\SQLEXPRESS;Initial
Catalog=DBUSUARIOS;Integrated Security=True");

     con.Open();

     string cod = CodigoCliente(usuario);

     int res = 0;
     string sql = "";

     if (cod == null)
          sql = "INSERT INTO Usuarios (NombreUsuario, CodigoC2DM) VALUES (@
usuario, @codigo)";
     else
          sql = "UPDATE Usuarios SET CodigoC2DM = @codigo WHERE NombreUsuario
= @usuario";

     SqlCommand cmd = new SqlCommand(sql, con);

     cmd.Parameters
           .Add("@usuario", System.Data.SqlDbType.NVarChar).Value = usuario;
     cmd.Parameters
           .Add("@codigo", System.Data.SqlDbType.NVarChar).Value = regGCM;

     res = cmd.ExecuteNonQuery();

     con.Close();

     return res;
}
```

El código es sencillo, pero ¿por qué es necesario considerar el caso del UPDATE? Como ya advertimos en el apartado anterior, el servidor GCM puede en ocasiones refrescar (actualizar) el ID de registro de un cliente comunicándoselo de nuevo a éste, por lo que a su vez la aplicación cliente tendrá que hacer también la misma actualización contra la aplicación web. Para ello, el cliente simplemente volverá a llamar al método `RegistroCliente()` del servicio web pasando el mismo nombre de usuario pero con el ID de registro actualizado. Para saber si el cliente está ya registrado o no el método se apoya en un método auxiliar llamado `CodigoCliente()` que realiza una búsqueda de un nombre de usuario en la base de datos para devolver su ID de registro en caso de encontrarlo. El código de este método es igual de sencillo que el anterior:

```
public string CodigoCliente(string usuario)
{
    SqlConnection con =
        new SqlConnection(
            @"Data Source=SGOLIVERPC\SQLEXPRESS;Initial
Catalog=DBUSUARIOS;Integrated Security=True");

    con.Open();

    string sql =
           "SELECT CodigoC2DM FROM Usuarios WHERE NombreUsuario = @usuario";

    SqlCommand cmd = new SqlCommand(sql, con);

    cmd.Parameters
           .Add("@usuario", System.Data.SqlDbType.NVarChar).Value = usuario;

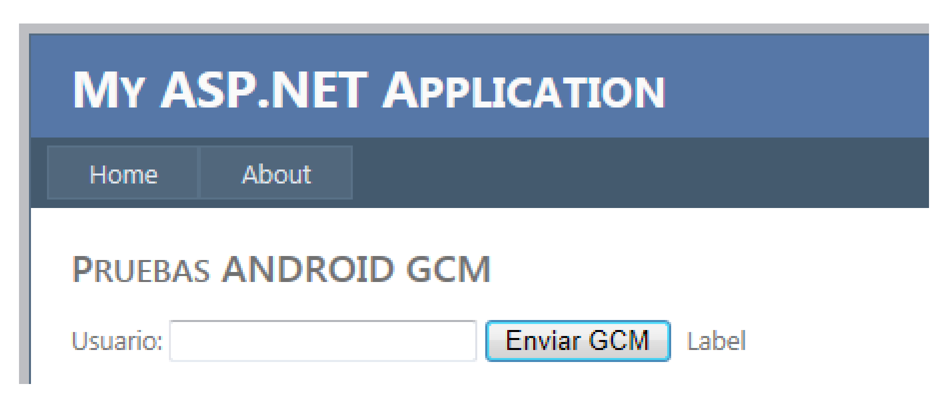
    string cod = (String)cmd.ExecuteScalar();

    con.Close();

    return cod;
}
```

Con esto ya tendríamos implementado nuestro servicio web para el registro de clientes.

Para el envío de los mensajes utilizaremos directamente la página "Default.aspx" creada por defecto al generar el proyecto de Visual Studio. Modificaremos esta página para añadir tan sólo un cuadro de texto donde podamos introducir el nombre de usuario asociado al cliente al que queremos enviar un mensaje, un botón "Enviar" con el realizar el envío, y una etiqueta donde mostrar el estado del envío. Quedaría algo como lo siguiente:



El botón de envío, realizará una búsqueda en la base de datos del nombre de usuario introducido, recuperará su *Registration ID* y enviará un mensaje de prueba con la fecha-hora actual.

```
protected void Button3_Click(object sender, EventArgs e)
{
     ServicioRegistroGCM svc = new ServicioRegistroGCM();

     string codUsuario = svc.CodigoCliente(TxtUsuario.Text);

     bool res = enviarMensajePrueba(codUsuario);

     if (res == true)
         LblResultadoMensaje.Text = "Envío OK";
     else
         LblResultadoMensaje.Text = "Envío NOK";
}
```

Como vemos en el código, toda la lógica de envío de mensajes la he encapsulado en el método auxiliar `enviarMensajePrueba()` para poder centrarme ahora en ella. En este método es donde vamos a hacer realmente uso de la API del servicio de *Google Cloud Messaging*, y por ello antes de ver la implementación vamos a hablar primero de las distintas opciones de esta API.

Todas las llamadas a la API de GCM para enviar mensajes se realizan mediante peticiones HTTP POST a la siguiente dirección:

`https://android.googleapis.com/gcm/send`

La cabecera de esta petición debe contener dos datos esenciales. Por un lado debemos indicar la API Key que generamos en el primer apartado (atributo `Authorization`), y por otro lado el formato del contenido (en este caso, los parámetros de la API) que vamos a incluir con la petición (atributo `Content-Type`). GCM permite formatear los datos como JSON (para lo que habría que indicar el valor "`application/json`") o como texto plano (para lo que debemos utilizar el valor "`application/x-www-form-urlencoded`"). En nuestro caso de ejemplo utilizaremos la segunda opción.

Dado que hemos elegido la opción de texto plano, los distintos datos del contenido se formatearán como parámetros HTTP con el formato tradicional, es decir, tendremos que construir una cadena de la forma "`param1=valor1&param2=valor2&…`".

Entre los distintos datos que podemos incluir hay tan solo uno obligatorio, llamado `registration_id`, que debe contener el ID de registro del cliente al que se le va a enviar el mensaje. A parte de éste también podemos incluir los siguientes parámetros opcionales:

- `delay_while_idle`. Hace que el servidor de GCM no envíe el mensaje al dispositivo mientras éste no se encuentre activo.
- `time_to_live`. Indica el tiempo máximo que el mensaje puede permanecer en el servidor de GCM sin entregar mientras el dispositivo está offline. Por defecto 4 semanas. Si se especifica algún valor también habrá que incluir el parámetro siguiente, `collapse_key`.
- `collapse_key`. Éste lo explicaré con un ejemplo. Imaginad que activamos el parámetro `delay_while_idle` y que el dispositivo que debe recibir el mensaje permanece inactivo varias horas. Si durante esas horas se generaran varias notificaciones hacia el dispositivo, estos mensajes se irían acumulando en el servidor de GCM y cuando el dispositivo se activara le llegarían todos de golpe. Esto puede tener sentido si cada mensaje contiene información distinta y relevante, pero ¿y si los mensajes simplemente fueran por ejemplo para decirle al dispositivo "Tienes correo nuevo"? Sería absurdo entregar en él varias notificaciones de este tipo en el mismo instante. Pues bien, para esto se utiliza el parámetro `collapse_key`. A este parámetro podemos asignarle como valor cualquier cadena de caracteres, de forma que si se acumulan en el servidor de GCM varios mensajes para el mismo dispositivo y con la misma `collapse_key`, al dispositivo sólo se le entregará el último de ellos cuando éste se active, descartando todos los demás.

- `data.<nombre_dato>`. Se pueden incluir tantos parámetros de este tipo como queramos, para incluir cualquier otra información que queramos en el mensaje. Por ejemplo podríamos pasar los datos de un nuevo correo recibido con dos parámetros como los siguientes: "data.emisor=aaa@gmail.com", y "data.asunto=pruebagcm". Tan solo recordad preceder el nombre de los datos con el prefijo "data.".

Una vez formateada convenientemente la cabecera y contenido de la petición HTTP, y realizada ésta a la dirección indicada anteriormente, podemos obtener diferentes respuestas dependiendo del resultado de la petición. Diferenciaremos los distintos resultados por el código de estado HTTP recibido en la respuesta:

- `200`. El mensaje se ha procesado correctamente, en cuyo caso se devuelve en los datos un parámetro "id=" con el código del mensaje generado.
- `401`. Ha fallado la autenticación de nuestra aplicación web contra los servidores de GCM. Normalmente significará algún problema con la API Key utilizada.
- `500`. Se ha producido un error al procesarse el mensaje. En este caso la respuesta incluirá en su contenido un parámetro "Error=" que indicará el código de error concreto devuelto por GCM.
- `501`. El servidor de GCM no está disponible temporalmente.

Y eso es todo, largo de contar pero sencillo en el fondo. Veamos cómo podemos implementar esto en C#, y para ello vamos a ver el código del método que dejamos antes pendiente, `enviarMensajePrueba()`, y justo después lo comentamos.

```
private static bool enviarMensajePrueba(String registration_id)
{
    String GCM_URL = @"https://android.googleapis.com/gcm/send";

    string collapseKey = DateTime.Now.ToString();

    Dictionary data = new Dictionary();
    data.Add("data.msg",
        HttpUtility.UrlEncode(
             "Prueba. Timestamp: " + DateTime.Now.ToString()));

    bool flag = false;
    StringBuilder sb = new StringBuilder();

    sb.AppendFormat("registration_id={0}&collapse_key={1}",
        registration_id, collapseKey);

    foreach (string item in data.Keys)
    {
        if (item.Contains("data."))
            sb.AppendFormat("&{0}={1}", item, data[item]);
    }

    string msg = sb.ToString();
    HttpWebRequest req = (HttpWebRequest)WebRequest.Create(GCM_URL);
    req.Method = "POST";
    req.ContentLength = msg.Length;
    req.ContentType = "application/x-www-form-urlencoded";

    string apiKey = "AIzaSyCJ7QSQAznAmhDzNTLSUE6uX9aUfr9-9RI";
    req.Headers.Add("Authorization:key=" + apiKey);
```

```
    using (StreamWriter oWriter = new StreamWriter(req.GetRequestStream()))
    {
        oWriter.Write(msg);
    }

    using (HttpWebResponse resp = (HttpWebResponse)req.GetResponse())
    {
        using (StreamReader sr = new StreamReader(resp.GetResponseStream()))
        {
            string respData = sr.ReadToEnd();

            if (resp.StatusCode == HttpStatusCode.OK)   // OK = 200
            {
                if (respData.StartsWith("id="))
                    flag = true;
            }
            else if (resp.StatusCode == HttpStatusCode.InternalServerError)
// 500
                Console.WriteLine(
                  "Error interno del servidor, prueba más tarde.");
            else if (resp.StatusCode == HttpStatusCode.ServiceUnavailable)
// 503
                Console.WriteLine(
                  "Servidor no disponible temporalmente, prueba más tarde.");
            else if (resp.StatusCode == HttpStatusCode.Unauthorized)
// 401
                Console.WriteLine("La API Key utilizada no es válida.");
            else
                Console.WriteLine("Error: " + resp.StatusCode);
        }
    }

    return flag;
}
```

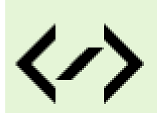
Como vemos el método recibe directamente como parámetro el *Registration ID* del cliente al que se va a enviar el mensaje. En primer lugar configuro todos los parámetros que pasará en la llamada a la API, que en este caso de ejemplo tan sólo serán, además del `registration_id` ya comentado, el `colapse_key`, y una dato adicional que llamaré "`data.msg`" (recordemos el prefijo "`data.`" obligatorio para este tipo de datos adicionales) con un mensaje de prueba que contenga la fecha/hora actual. Toda la cadena con estos parámetros la construyo utilizando un objeto `StringBuilder`. Lo único reseñable hasta ahora sería la forma de añadir el parámetro adicional data.msg, que lo hago mediante la creación de un objeto `Dictionary` y su método `add()` para añadir el dato, para poco después generar la cadena final recorriendo este diccionario en un bucle foreach. En este caso no sería necesaria toda esta parafernalia dado que sólo vamos a añadir un dato adicional, pero lo he dejado así para que tengáis un ejemplo de "patrón" mediante el cual podeis añadir más de un dato adicional de una forma sencilla y organizada.

Una vez creada la cadena de parámetros y datos que incluiremos como contenido de la petición creamos dicha petición como un objeto `HttpWebRequest` indicando la URL del servicio. Indicamos que la petición será de tipo POST asignando la propiedad `Method`, y configuramos la cabecera con los dos datos que ya hemos comentado antes (`Authorization` y `Content-Type`). El primero de ellos al ser "personalizado" debemos añadirlo utilizando el método `Add()` de la colección `Headers` de la petición. En cambio para el segundo existe una propiedad del objeto `HttpWebRequest` con la que podemos establecerlo directamente, llamada `ContentType`. Hecho esto, tan sólo nos queda añadir el contenido a la petición, lo que conseguimos obteniendo el stream de escritura de la petición mediante `GetRequestStream()` y escribiendo en él nuestra cadena de parámetros mediante el método `Write()`.

Seguidamente vamos a ejecutar la petición y a obtener la respuesta como objeto `HttpWebResponse`

mediante una llamada a `GetResponse()`. Por último, obtenemos el código de estado HTTP de la respuesta mediante la consulta a su propiedad `StatusCode`, y los datos asociados obteniendo el stream de lectura de la respuesta mediante `GetResponseStream()` y el método `ReadToEnd()` para leer todo el contenido. Evaluando estos dos datos determinamos fácilmente el resultado del envío según la información ya comentada antes en el texto.

Y con esto habríamos terminado la implementación del servidor. Haremos las pruebas pertinentes y mostraré el resultado cuando implementemos la aplicación Android cliente en el próximo apartado.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/dotnet-gcmserver

Implementación del Cliente Android

En los dos apartados anteriores hemos hablado sobre el servicio Google Cloud Messaging y hemos visto como implementar una aplicación web que haga uso de dicho servicio para enviar mensajes a dispositivos Android. Para cerrar el círculo, en este nuevo apartado nos centraremos en la aplicación Android cliente.

Esta aplicación cliente, como ya hemos comentado en alguna ocasión será responsable de:

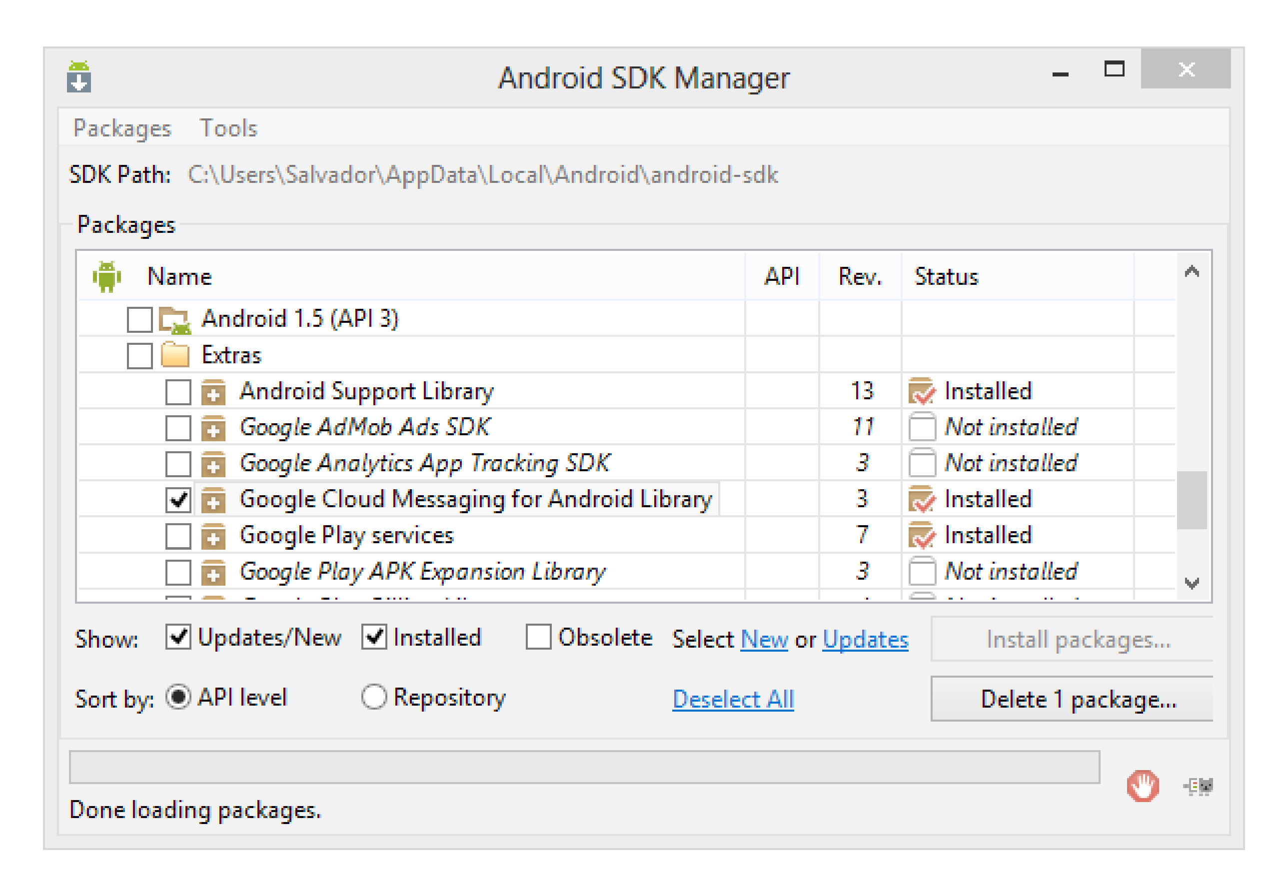
- Registrarse contra los servidores de GCM como cliente capaz de recibir mensajes.
- Almacenar el "Registration ID" recibido como resultado del registro anterior.
- Comunicar a la aplicación web el "Registration ID" de forma que ésta pueda enviarle mensajes.
- Recibir y procesar los mensajes desde el servidor de GCM.

Para las tareas 1 y 4 utilizaremos una librería específica de GCM que nos proporciona Google para facilitarnos la vida como desarrolladores (es posible hacerlo sin el uso de librerías externas, aunque requiere más trabajo). El punto 2 lo resolveremos fácilmente mediante el uso de `SharedPreferences`. Y por último el punto 3 lo implementaremos mediante la conexión al servicio web SOAP que creamos en el apartado anterior, sirviéndonos para ello de la librería ksoap2, tal como ya describimos en el capítulo sobre servicios web SOAP en Android.

Durante el capítulo construiremos una aplicación de ejemplo muy sencilla, en la que el usuario podrá introducir un nombre de usuario identificativo y pulsar un botón para que quede guardado en las preferencias de la aplicación. Tras esto podrá registrarse como cliente capaz de recibir mensajes desde GCM pulsando un botón llamado "Registrar GCM". En caso de realizarse de forma correcta este registro la aplicación enviará automáticamente el *Registration ID* recibido y el nombre de usuario almacenado a la aplicación web a través del servicio web. Igualmente el usuario podrá des-registrarse en el servicio GCM para no recibir más mensajes pulsando un botón llamado "Des-registrar GCM". Obviamente todo este proceso de registro y des-registro debería hacerse de forma transparente para el usuario de una aplicación real, en esta ocasión he colocado botones para ello sólo por motivos didácticos.

Antes de nada vamos a preparar nuestro proyecto de Eclipse y vamos a configurar convenientemente el AndroidManifest para poder hacer uso del servicio GCM y su librería auxiliar.

Para ello vamos a crear un nuevo proyecto Android sobre un target de Android 2.2 o superior que incluya las librerías de Google, y vamos a incluir en su carpeta /libs las librerías de ksoap2 (esto ya vimos como hacerlo en el capítulo sobre servicios web) y la librería cliente de GCM llamada "gcm.jar". ¿Cómo podemos obtener esta librería? Para conseguirla debemos instalar desde el Android SDK Manager el paquete extra llamado "*Google Cloud Messaging for Android Library*".



Una vez instalado podemos ir a la ruta "RAIZ_SDK_ANDROID/extras/google/gcm/gcm-client/dist" donde deberá aparecer la librería "gcm.jar" que debemos añadir a nuestro proyecto.

Lo siguiente será configurar nuestro AndroidManifest. Lo primero que añadiremos será una cláusula <uses-sdk> para indicar como versión mínima del SDK soportada la 8 (Android 2.2). Con esto nos aseguraremos de que la aplicación no se instala en dispositivos con versión de Android anterior, no soportadas por el servicio GCM.

```
<uses-sdk
     android:minSdkVersion="8"
     android:targetSdkVersion="17" />
```

A continuación añadiremos los permisos necesarios para ejecutar la aplicación y utilizar GCM:

```
<permission
     android:name="net.sgoliver.android.gcm.permission.C2D_MESSAGE"
     android:protectionLevel="signature" />

<uses-permission
        android:name="net.sgoliver.android.gcm.permission.C2D_MESSAGE" />

<uses-permission android:name="com.google.android.c2dm.permission.RECEIVE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
```

Los dos primeros aseguran que sólo esta aplicación podrá recibir los mensajes, el segundo permite la recepción en sí de mensajes desde GCM (sustituir mi paquete java "`net.sgoliver.android.gcm`" por el vuestro propio en estas lineas), el tercero es el permiso para poder conectarnos a internet y el último es necesario para tareas realizadas durante la recepción de mensajes que veremos más adelante.
Por último, como componentes de la aplicación, además de la actividad principal ya añadida por defecto, deberemos declarar un *broadcast receiver* llamado `GCMBroadcastReceiver`, que no tendremos que

crearlo porque ya viene implementado dentro de la librería "gcm.jar" (solo tenéis que modificar el elemento `<category>` para indicar vuestro paquete java), y un servicio llamado `GCMIntentService` (Atención, es obligatorio este nombre exacto para el servicio si no queremos tener que implementar nosotros mismos el broadcast receiver anterior). Ya veremos más adelante para qué son estos dos componentes.

```
<application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

    ...

    <receiver android:name="com.google.android.gcm.GCMBroadcastReceiver"
        android:permission="com.google.android.c2dm.permission.SEND" >
        <intent-filter>
         <action android:name="com.google.android.c2dm.intent.RECEIVE" />
         <action android:name="com.google.android.c2dm.intent.REGISTRATION" />
         <category android:name="net.sgoliver.android.gcm" />
        </intent-filter>
    </receiver>

    <service android:name=".GCMIntentService" />

</application>
```

Una vez definido nuestro AndroidManifest con todos los elementos necesarios vamos a empezar a implementar la funcionalidad de nuestra aplicación de ejemplo.

Empezamos por la más sencilla, el botón de guardar el nombre de usuario. Como comentamos anteriormente, vamos a utilizar preferencias compartidas para esta tarea. Como éste es un tema ya visto en el curso no me detendré en el código ya que es bastante directo.

```
btnGuardarUsuario.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        SharedPreferences prefs =
             getSharedPreferences("MisPreferencias", Context.MODE_PRIVATE);

        SharedPreferences.Editor editor = prefs.edit();
        editor.putString("usuario", txtUsuario.getText().toString());
        editor.commit();
    }
});
```

Como podéis comprobar nos limitamos a almacenar una nueva propiedad llamada "usuario" con el texto introducido en el cuadro de texto de la interfaz.

El siguiente botón es el de registro del cliente en GCM, y aquí sí nos detendremos un poco para comentar primero cómo funciona internamente este procedimiento.

La aplicación android debe solicitar el registro en el servicio GCM mediante una petición HTTP POST similar a la que ya vimos en el apartado anterior para la aplicación web. Por suerte, este procedimiento se ve simplificado enormemente con el uso de la librería gcm.jar, ya que el montaje y ejecución de esta petición queda encapsulado como una simple llamada a un método estático de la clase `GCMRegistrar`, definida en la librería. Por su parte, tanto la respuesta a esta petición de registro como la posterior recepción de mensajes se reciben en la aplicación Android en forma de intents. Y aquí es donde entran en juego los dos componentes que hemos definido anteriormente en nuestro AndroidManifest. El receiver

GCMBroadcastReceiver será el encargado de "esperar y capturar" estos intents cuando se reciban y posteriormente lanzar el servicio GCMIntentService donde se procesarán en un hilo independiente estas respuestas según el intent recibido. Como ya dijimos, el broadcast receiver no será necesario crearlo ya que utilizaremos el ya proporcionado por la librería. En cambio la implementación del IntentService sí será de nuestra responsabilidad. Aunque una vez más la librería de GCM nos facilitará esta tarea como ya veremos más adelante.

Veamos primero cómo realizar el registro de nuestra aplicación en GCM al pulsar el botón de registro. Como hemos dicho esto se limitará a llamar a un método estático, llamado register(), de la clase GCMRegistrar. La única precaución que tomaremos es verificar previamente si estamos ya registrados, algo que podremos hacer fácilmente llamando al método getRegistrationId() de la misma clase. El método register() recibirá dos parámetros, el primero de ellos una referencia al contexto de la aplicación (normalmente la actividad desde la que se llama) y en segundo lugar el "Sender ID" que obtuvimos cuando creamos el nuevo proyecto en la Google API Console.

```
btnRegistrar.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {

        //Si no estamos registrados --> Nos registramos en GCM
        final String regId = GCMRegistrar.getRegistrationId(GcmActivity.this);

        if (regId.equals("")) {
            //Sender ID
             GCMRegistrar.register(GcmActivity.this, "224338875065");
        } else {
            Log.v("GCMTest", "Ya registrado");
        }
    }
});
```

Así de sencillo y rápido. Pero esto es sólo la petición de registro, ahora nos tocará esperar la respuesta, algo que veremos en breve.

Por su parte, el botón de "des-registro" se implementará de forma análoga, con la única diferencia que esta vez utilizaremos el método unregister() de la clase GCMRegistrar.

```
btnDesRegistrar.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {

        //Si estamos registrados --> Nos des-registramos en GCM
        final String regId = GCMRegistrar.getRegistrationId(GcmActivity.this);
        if (!regId.equals("")) {
            GCMRegistrar.unregister(GcmActivity.this);
        } else {
            Log.v("GCMTest", "Ya des-registrado");
        }
    }
});
```

Ahora toca procesar las respuestas. Como hemos dicho, para hacer esto tendremos que implementar el servicio GCMIntentService. Pero no lo haremos desde cero, ya que la librería de GCM nos proporciona una clase base GCMBaseIntentService de la cual podemos extender la nuestra, con la ventaja de que tan sólo tendremos que sobrescribir unos pocos métodos a modo de callbacks, uno por cada posible respuesta o mensaje que podemos recibir desde el servicio GCM.

Los métodos son los siguientes:

- `onRegistered(context, regId)`. Se llamará al recibirse una respuesta correcta a la petición de registro e incluye como parámetro el Registration ID asignado a nuestro cliente.
- `onUnregistered(context, regId)`. Análogo al anterior pero aplicado a una petición de "des-registro".
- `onError(context, errorId)`. Se llamará al recibirse una respuesta de error a una petición de registro o des-registro. El código de error concreto se recibe como parámetro.
- `onMessage(context, intent)`. Se llamará cada vez que se reciba un nuevo mensaje desde el servidor de GCM. El contenido del mensaje se recibe en forma de intent, el cual veremos más adelante cómo procesar.

Empecemos por el método `onRegistered()`. Al recibir una respuesta satisfactoria a la petición de registro recuperaremos el nombre de usuario almacenado y junto con el Registration ID recibido nos conectaremos al servicio web que creamos en el apartado pasado pasándole dichos datos. Esto completará el registro tanto con el servidor de GCM como con nuestra aplicación web.

```
protected void onRegistered(Context context, String regId) {
    Log.d("GCMTest", "REGISTRATION: Registrado OK.");

    SharedPreferences prefs =
       context.getSharedPreferences("MisPreferencias", Context.MODE_PRIVATE);

    String usuario = prefs.getString("usuario", "por_defecto");

    registroServidor(usuario, regId);
}
```

El método `registroServidor()` será el encargado de realizar la conexión al servicio web y de la llamada al método web de registro. No me detendré en comentar el código de este método porque es análogo a los ejemplos ya vistos en el capítulo que dedicamos a servicios web SOAP en Android. Veamos tan sólo el código:

```
private void registroServidor(String usuario, String regId)
{
    final String NAMESPACE = "http://sgoliver.net/";
    final String URL="http://10.0.2.2:1634/ServicioRegistroGCM.asmx";
    final String METHOD_NAME = "RegistroCliente";
    final String SOAP_ACTION = "http://sgoliver.net/RegistroCliente";

    SoapObject request = new SoapObject(NAMESPACE, METHOD_NAME);

    request.addProperty("usuario", usuario);
    request.addProperty("regGCM", regId);

    SoapSerializationEnvelope envelope =
        new SoapSerializationEnvelope(SoapEnvelope.VER11);

    envelope.dotNet = true;

    envelope.setOutputSoapObject(request);

    HttpTransportSE transporte = new HttpTransportSE(URL);
```

```
    try
    {
        transporte.call(SOAP_ACTION, envelope);

        SoapPrimitive resultado_xml =(SoapPrimitive)envelope.getResponse();
        String res = resultado_xml.toString();

        if(res.equals("1"))
            Log.d("GCMTest", "Registro WS: OK.");
    }
    catch (Exception e)
    {
        Log.d("GCMTest", "Registro WS: NOK. " + e.getCause() + " || " +
e.getMessage());
    }
}
```

Por su parte, en los métodos de des-registro y de error me limitaré a escribir un mensaje en el log de la aplicación para no complicar demasiado el ejemplo, pero en una aplicación real deberíamos verificar estas respuestas.

```
@Override
protected void onUnregistered(Context context, String regId) {
    Log.d("GCMTest", "REGISTRATION: Desregistrado OK.");
}

@Override
protected void onError(Context context, String errorId) {
    Log.d("GCMTest", "REGISTRATION: Error -> " + errorId);
}
```

Por último, en el método `onMessage()` procesaremos el intent con los datos recibidos en el mensaje y mostraremos una notificación en la barra de estado de Android. El *intent* recibido contendrá un elemento en su colección de extras por cada dato adicional que se haya incluido en la petición que hizo la aplicación servidor al enviar el mensaje. ¿Recordáis? Aquellos datos adicionales que había que preceder con el prefijo "`data.`". Si hacéis memoria, en nuestros mensajes de ejemplo tan sólo incluíamos un dato llamado "`data.msg`" con un mensaje de prueba. Pues bien, estos datos se recuperarán de la colección de extras del intent llamado al método `getString()` con el nombre del dato, pero esta vez eliminando el prefijo "`data.`". Veamos cómo quedaría todo esto:

```
@Override
protected void onMessage(Context context, Intent intent) {
    String msg = intent.getExtras().getString("msg");
    Log.d("GCMTest", "Mensaje: " + msg);
    mostrarNotificacion(context, msg);
}
```

Simple, ¿no?. Al final del método llamamos a un método auxiliar `mostrarNotificacion()` que será el encargado de mostrar la notificación en la barra de estado de Android. Esto también vimos como hacerlo en detalle en un apartado anterior por lo que tampoco comentaremos el código:

```
private void mostrarNotificacion(Context context, String msg)
{
    //Obtenemos una referencia al servicio de notificaciones
    String ns = Context.NOTIFICATION_SERVICE;
    NotificationManager notManager =
        (NotificationManager) context.getSystemService(ns);

    //Configuramos la notificación
    int icono = android.R.drawable.stat_sys_warning;
    CharSequence textoEstado = "Alerta!";
    long hora = System.currentTimeMillis();

    Notification notif =
        new Notification(icono, textoEstado, hora);

    //Configuramos el Intent
    Context contexto = context.getApplicationContext();
    CharSequence titulo = "Nuevo Mensaje";
    CharSequence descripcion = msg;

    Intent notIntent = new Intent(contexto,
        GCMIntentService.class);

    PendingIntent contIntent = PendingIntent.getActivity(
        contexto, 0, notIntent, 0);

    notif.setLatestEventInfo(
        contexto, titulo, descripcion, contIntent);

    //AutoCancel: cuando se pulsa la notificaión ésta desaparece
    notif.flags |= Notification.FLAG_AUTO_CANCEL;

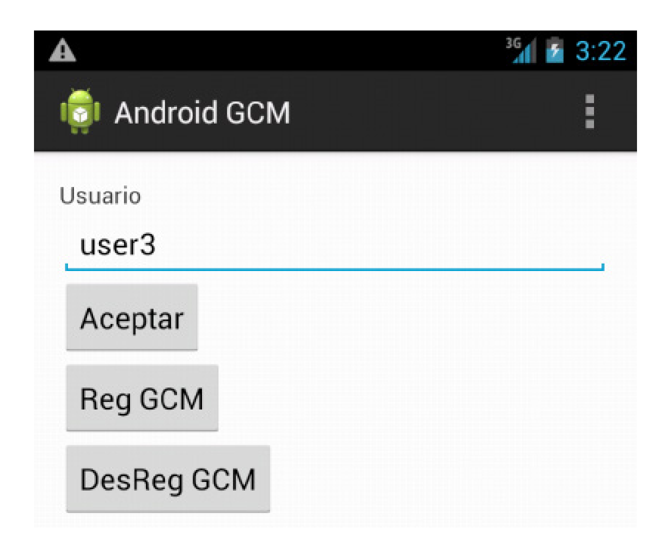
    //Enviar notificación
    notManager.notify(1, notif);
}
```

Y sólo una indicación más, además de sobrescribir estos métodos en nuestra clase `GCMIntentService`, también tendremos que añadir un nuevo constructor sin parámetros que llame directamente al constructor de la clase base pasándole de nuevo el Sender ID que obtuvimos al crear el nuevo proyecto en la Google API Console. Quedaría algo así:

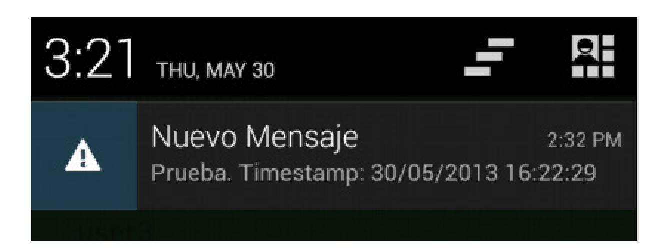
```
public GCMIntentService() {
    super("224338875065");
}
```

Si no ha quedado claro del todo cómo quedaría la clase `GCMIntentService` completa puede descargarse y consultarse el código fuente completo al final del capítulo.

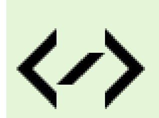
Y con esto habríamos terminado de implementar nuestra aplicación Android capaz de recibir mensajes push desde nuestra aplicación web de ejemplo. Si ejecutamos ambas y todo ha ido bien, introducimos un nombre de usuario en la aplicación Android, pulsamos "Aceptar" para guardarlo, nos registramos como clientes en GCM pulsando el botón "Registrar GCM", y seguidamente desde la aplicación web introducimos el mismo nombre de usuario del cliente, pulsamos el botón "Enviar GCM" y en breves segundos nos debería aparecer la notificación en la barra de estado de nuestro emulador como se observa en las imágenes siguientes:



Y si desplegamos la barra de estado veremos el mensaje de prueba recibido:



Como habéis podido comprobar en estos tres últimos capítulos, la utilización de mensajes push requiere de un proceso algo laborioso pero para nada complicado. Os animo a que lo intentéis en vuestras aplicaciones ya que puede representar una característica interesante y útil.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-gcm

16

Depuración

XVI. Depuración en Android

Logging en Android

Hacemos un pequeño alto en el camino en el Curso de Programación Android para hablar de un tema que, si bien no es específico de Android, sí nos va a resultar bastante útil a la hora de explorar otras características de la plataforma.

Una de las técnicas más útiles a la hora de depurar y/o realizar el seguimiento de aplicaciones sobre cualquier plataforma es la creación de logs de ejecución. Android por supuesto no se queda atrás y nos proporciona también su propio servicio y API de *logging* a través de la clase `android.util.Log`.

En Android, todos los mensajes de log llevarán asociada la siguiente información:

- Fecha/Hora del mensaje.
- Criticidad. Nivel de gravedad del mensaje (se detalla más adelante).
- PID. Código interno del proceso que ha introducido el mensaje.
- Tag. Etiqueta identificativa del mensaje (se detalla más adelante).
- Mensaje. El texto completo del mensaje.

De forma similar a como ocurre con otros frameworks de logging, en Android los mensajes de log se van a clasificar por su criticidad, existiendo así varias categorias (ordenadas de mayor a menor criticidad):

1. Error
2. Warning
3. Info
4. Debug
5. Verbose

Para cada uno de estos tipos de mensaje existe un método estático independiente que permite añadirlo al log de la aplicación. Así, para cada una de las categorías anteriores tenemos disponibles los métodos `e()`, `w()`, `i()`, `d()` y `v()` respectivamente.

Cada uno de estos métodos recibe como parámetros la etiqueta (*tag*) y el texto en sí del mensaje. Como etiqueta de los mensajes, aunque es un campo al que podemos pasar cualquier valor, suele utilizarse el nombre de la aplicación o de la actividad concreta que genera el mensaje. Esto nos permitirá más tarde crear filtros personalizados para identificar y poder visualizar únicamente los mensajes de log que nos interesan, entre todos los generados por Android [que son muchos] durante la ejecución de la aplicación.

Hagamos un miniprograma de ejemplo para ver cómo fuenciona esto. El programa será tan simple como añadir varios mensajes de log dentro del mismo onCreate de la actividad principal y ver qué ocurre. Os muestro el código completo:

```
public class LogsAndroid extends Activity {

    private static final String LOGTAG = "LogsAndroid";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        Log.e(LOGTAG, "Mensaje de error");
        Log.w(LOGTAG, "Mensaje de warning");
        Log.i(LOGTAG, "Mensaje de información");
        Log.d(LOGTAG, "Mensaje de depuración");
        Log.v(LOGTAG, "Mensaje de verbose");
    }
}
```

Si ejecutamos la aplicación anterior en el emulador veremos cómo se abre la pantalla principal que crea Eclipse por defecto y aparentemente no ocurre nada más. ¿Dónde podemos ver los mensajes que hemos añadido al log? Pues para ver los mensajes de log nos tenemos que ir a la perspectiva de Eclipse llamada *DDMS*. Una vez en esta perspectiva, podemos acceder a los mensajes de log en la parte inferior de la pantalla, en una vista llamada *LogCat*. En esta ventana se muestran todos los mensajes de log que genera Android durante la ejecución de la aplicación, que son muchos, pero si buscamos un poco en la lista encontraremos los generados por nuestra aplicación.

Search for messages. Accepts Java regexes. Prefix with pid:, app:, tag: or text: to limit scope.

Level	Time	PID	TID	Application	Tag	Text
W	05-27 19:36:32.261	1353	1353	com.android.inputmethod.l...	Trace	Unexpected value from nativeGetEnabledTags: 0
W	05-27 19:36:32.261	1353	1353	com.android.inputmethod.l...	Trace	Unexpected value from nativeGetEnabledTags: 0
E	05-27 19:36:32.268	803	1025		AudioFlinger	no more track names available
E	05-27 19:36:32.268	1227	1289	system_process	AudioTrack	AudioFlinger could not create track, status: -12
E	05-27 19:36:32.268	1227	1289	system_process	SoundPool	Error creating AudioTrack
D	05-27 19:36:32.311	1384	1384	com.android.launcher	dalvikvm	GC_FOR_ALLOC freed 343K, 17% free 7119K/8564K, pa s
I	05-27 19:36:32.311	1384	1384	com.android.launcher	dalvikvm-heap	Grow heap (frag case) to 9.884MB for 2960652-byte
D	05-27 19:36:32.328	1384	1393	com.android.launcher	dalvikvm	GC_FOR_ALLOC freed <1K, 13% free 10010K/11456K, p

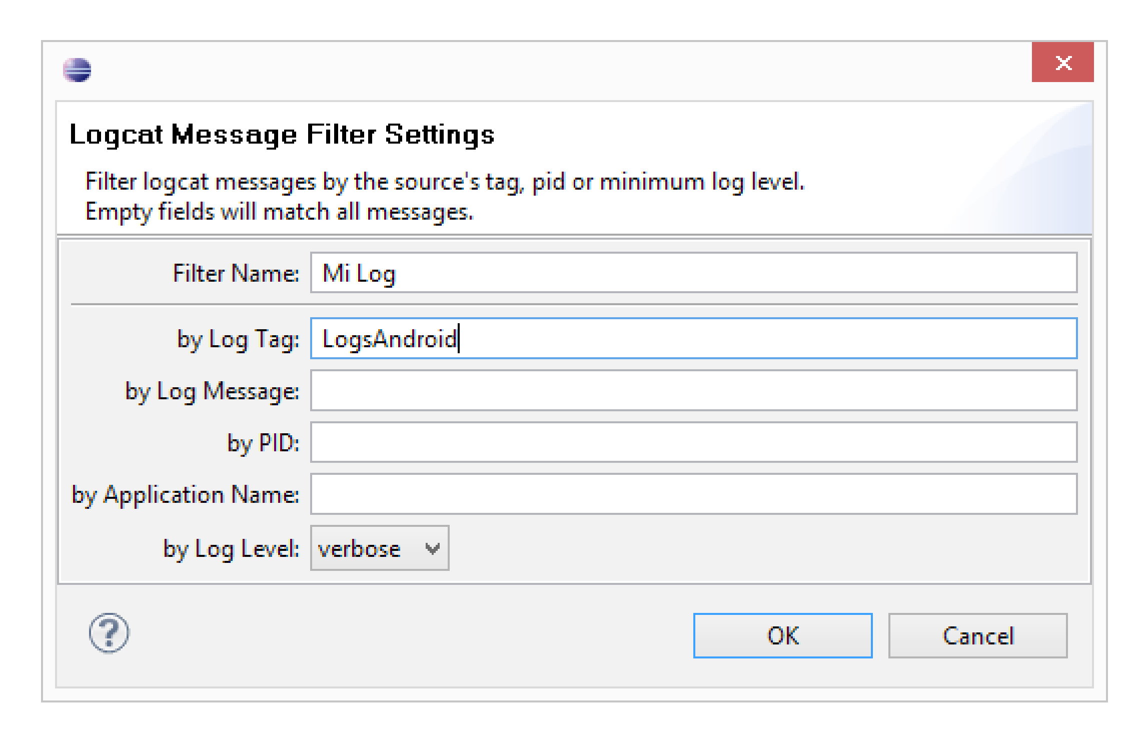
Como se puede observar, para cada mensaje se muestra toda la información que indicamos al principio del apartado, además de estar diferenciados por un color distinto según su criticidad.

En este caso de ejemplo, los mensajes mostrados son pocos y fáciles de localizar en el log, pero para una aplicación real, el número de estos mensajes puede ser mucho mayor y aparecer intercalados caóticamente entre los demás mensajes de Android. Para estos casos, la ventada de LogCat ofrece una serie de funcionalidades para facilitar la visualización y búsqueda de determinados mensajes.

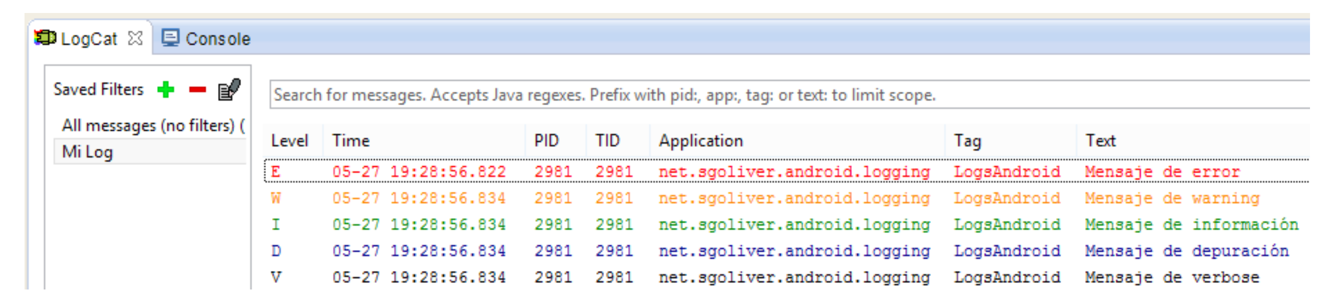
Por ejemplo, podemos restringir la lista para que sólo muestre mensajes con una determinada criticidad mínima. Esto se consigue pulsando alguno de los 5 primeros botones que se observan en la parte superior derecha de la ventana de log. Así, si por ejemplo pulsamos sobre el botón de la categoría *Info* (en verde), en la lista sólo se mostrarán los mensajes con criticidad *Error*, *Warning* e *Info*.

Otro método de filtrado más interesante es la definición de filtros personalizados (botón "+" verde), donde podemos filtrar la lista para mostrar únicamente los mensajes con un PID o Tag determinado. Si hemos utilizado como etiqueta de los mensajes el nombre de nuestra aplicación o de nuestras actividades esto nos proporcionará una forma sencilla de visualizar sólo los mensajes generados por nuestra aplicación.

Así, para nuestro ejemplo, podríamos crear un filtro indicando como Tag la cadena "*LogsAndroid*", tal como se muestra en la siguiente imagen:



Esto creará una nueva ventana de log con el nombre que hayamos especificado en el filtro, donde sólo aparecerán nuestros 5 mensajes de log de ejemplo:



Por último, cabe mencionar que existe una variante de cada uno de los métodos de la clase Log que recibe un parámetro más en el que podemos pasar un objeto de tipo excepción. Con esto conseguimos que, además del mensaje de log indicado, se muestre también la traza de error generada con la excepción.

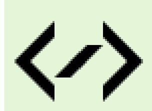
Veamos esto con un ejemplo, y para ello vamos a forzar un error de división por cero, vamos a capturar la excepción y vamos a generar un mensaje de log con la variante indicada:

```
try
{
    int a = 1/0;
}
catch(Exception ex)
{
    Log.e(LOGTAG, "División por cero!", ex);
}
```

Si volvemos a ejecutar la aplicación y vemos el log generado, podremos comprobar cómo se muestra la traza de error correspondiente generada con la excepción.

Level	Time	PID	TID	Application	Tag	Text
E	05-27 19:28:56.822	2981	2981	net.sgoliver.android.logging	LogsAndroid	Mensaje de error
W	05-27 19:28:56.834	2981	2981	net.sgoliver.android.logging	LogsAndroid	Mensaje de warning
I	05-27 19:28:56.834	2981	2981	net.sgoliver.android.logging	LogsAndroid	Mensaje de información
D	05-27 19:28:56.834	2981	2981	net.sgoliver.android.logging	LogsAndroid	Mensaje de depuración
V	05-27 19:28:56.834	2981	2981	net.sgoliver.android.logging	LogsAndroid	Mensaje de verbose
E	05-27 19:28:56.834	2981	2981	net.sgoliver.android.logging	LogsAndroid	División por cero!
E	05-27 19:28:56.834	2981	2981	net.sgoliver.android.logging	LogsAndroid	java.lang.ArithmeticException: divide by zero
E	05-27 19:28:56.834	2981	2981	net.sgoliver.android.logging	LogsAndroid	at net.sgoliver.android.logging.MainActivity.on .java:25)
E	05-27 19:28:56.834	2981	2981	net.sgoliver.android.logging	LogsAndroid	at android.app.Activity.performCreate(Activity.

En definitiva, podemos comprobar como la generación de mensajes de log puede ser una herramienta sencilla pero muy efectiva a la hora de depurar aplicaciones que no se ajustan mucho a la depuración paso a paso, o simplemente para generar trazas de ejecución de nuestras aplicaciones para comprobar de una forma sencilla cómo se comportan.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

curso-android-src/android-logging